

ON IMPROVING EFFICIENCY OF DATA-INTENSIVE APPLICATIONS IN GEO-DISTRIBUTED
ENVIRONMENTS

Fan Jiang

A dissertation submitted to the faculty of the University of North Carolina at Chapel Hill in partial fulfillment
of the requirements for the degree of Doctor of Philosophy in the Department of Computer Science.

Chapel Hill
2020

Approved by:

Stan Ahalt

Claris Castillo

Jay Aikat

Shahriar Nirjon

Kapil Singh

© 2020
Fan Jiang
ALL RIGHTS RESERVED

ABSTRACT

Fan Jiang: On Improving Efficiency of Data-Intensive Applications in Geo-Distributed Environments
(Under the direction of Stan Ahalt and Claris Castillo)

Distributed systems are pervasively demanded and adopted in nowadays for processing data-intensive workloads since they greatly accelerate large-scale data processing with scalable parallelism and improved data locality. Traditional distributed systems initially targeted computing clusters but have since evolved to data centers with multiple clusters. These systems are mostly built on top of homogeneous, tightly integrated resources connected in high-speed local-area networks (LANs), and typically require data to be ingested to a central data center for processing. Today, with enormous volumes of data continuously generated from geographically distributed locations, direct adoption of such systems is prohibitively inefficient due to the limited system scalability and high cost for centralizing the geo-distributed data over the wide-area networks (WANs). More commonly, it becomes a trend to build geo-distributed systems wherein data processing jobs are performed on top of geo-distributed, heterogeneous resources in proximity to the data at vastly distributed geo-locations. However, critical challenges and mechanisms for efficient execution of data-intensive applications in such geo-distributed environments are unclear by far.

The goal of this dissertation is to identify such challenges and mechanisms, by extensively using the research principles and methodology of conventional distributed systems to investigate the geo-distributed environment, and by developing new techniques to tackle these challenges and run data-intensive applications with efficiency at scale. The contributions of this dissertation are threefold.

Firstly, the dissertation shows that the high level of resource heterogeneity exhibited in the geo-distributed environment undermines the scalability of geo-distributed systems. Virtualization-based resource abstraction mechanisms have been introduced to abstract the hardware, network, and OS resources throughout the system, to mitigate the underlying resource heterogeneity and enhance the system scalability.

Secondly, the dissertation reveals the overwhelming performance and monetary cost incurred by indulgent data sharing over the WANs in geo-distributed systems. Network optimization approaches, including linear-programming-based global optimization, greedy bin-packing heuristics, and TCP enhancement, are developed

to optimize the network resource utilization and circumvent unnecessary expenses imposed on data sharing in WANs.

Lastly, the dissertation highlights the importance of data locality for data-intensive applications running in the geo-distributed environment. Novel data caching and locality-aware scheduling techniques are devised to improve the data locality.

To Kobe Bryant

For leading my way to where I am and where I will be.

ACKNOWLEDGMENTS

First of all, I would like to thank my advisors Stan Ahalt and Claris Castillo for their unreserved support and help throughout my Ph.D. career. They not only provide guidance to me in my research, but also offer me valuable advice and help for my life. I started my Ph.D. career with little experience in research and have made every possible mistake one can make in the past five years, but they show great patience in mentoring me and always bring me back to the right track, which I am very grateful for. Stan and Claris are veteran researchers who can quickly pinpoint the crux and provide me useful feedback during our research discussions. I truly enjoy every discussion with them, from which I always learn enlightening ideas that resolve my confusions and help me overcome obstacles in research. They also give me a lot of guidance with my formal writing in English, which I was extremely struggling with at the beginning of my Ph.D. career. Furthermore, I also want to thank the members of my dissertation committee: Jay Aikat, Shahriar Nirjon, and Kapil Singh. I do appreciate all the advice and help from you, and have learned a lot from our discussions about and beyond this dissertation.

I also would like to thank the RENCI family for their wholehearted support. Notably, I want to thank Charles Schmitt for giving me the opportunity to work in RENCI and learn research. I also would like to thank Ray Idaszak and Steven Cox for their help and advice on the projects we have worked on together. I am also very thankful to the research and engineering teams at RENCI, including the ExoGENI Team, ACIS Team, and DevOps Team, for their dedicated support throughout my Ph.D. career. I want to specially thank Hong Yi, Michael Stealey, Kyle Ferriter, Chris Calloway, Ananya Mukherjee, Diptorup Deb, Marcin Sliwowski, Cole Dickens, Terrell Russell, Anirban Mandal, Paul Ruth, Mert Cevik, and Ilya Baldin.

Furthermore, I would like to express my gratitude to our collaborators on the research projects related to this dissertation: Alex Feltus, Paul Avillach, Chris Ball, Gregoire Versmee, and Laura Thesillat-Versmee. This dissertation is impossible to be completed without your contributions. I also want to thank Don Smith and Mohit Bansal for their excellent lectures, from which I have learned essential theories and skillset necessary for my dissertation. I am also thankful to staff members in the department and RENCI, notably Julie Hayes, Jeremy Zollars, Asia Mieczkowska, Jodie Gregoritsch, Denise Kenney, and Margie Wesley.

Finally, I would like to thank my family, particularly my parents, Jun, Charlie, and Le, for their unconditional support and love.

The research in this dissertation was supported by NSF grants, ACI 1440715, ACI 1659300, and NIH grant OT3 ODO25464-01.

TABLE OF CONTENTS

LIST OF TABLES	xiii
LIST OF FIGURES	xiv
LIST OF ABBREVIATIONS	xvi
1 Introduction	1
1.1 Data-Intensive Applications	1
1.1.1 Data Volume	1
1.1.2 Data Velocity	3
1.1.3 Data Variety	4
1.2 Geo-Distributed Computing	5
1.2.1 High Scalability	6
1.2.2 Low Latency	6
1.2.3 Strong Resilience	7
1.2.4 Regulatory Compliance	8
1.2.5 Cost Effectiveness	8
1.3 Challenges for Data-Intensive Applications in Geo-Distributed Environments	9
1.3.1 High Degree of Heterogeneity	9
1.3.2 Loss of Data Locality	10
1.3.3 Inefficiency in Data Sharing	11
1.4 Thesis Statement	12
1.5 Contributions	13
1.5.1 Resource Abstraction	13

1.5.2	Preservation of Data Locality	13
1.5.3	Network Optimization	14
1.6	Organization	15
2	Background and Prior Work	16
2.1	Virtualization-Based Resource Abstraction	16
2.1.1	Hardware Virtualization	16
2.1.2	OS-Level Virtualization	17
2.1.3	Network Virtualization	20
2.2	Data Locality in Distributed Computing	22
2.2.1	Computation Caching	23
2.2.2	Distributed Data Caching and Replication	23
2.2.3	Locality-Aware Scheduling	25
2.2.4	Data Locality and Cost Awareness in the Cloud	27
2.3	TCP Optimization for Remote Data Transfer	28
3	Advancing Data-Driven Scientific Collaboration in the Geo-Distributed Environment	31
3.1	Collaboration Representation	32
3.1.1	Scope	32
3.1.2	Collaboration Model	32
3.1.2.1	Collaborator	32
3.1.2.2	Data Policy	33
3.1.2.3	Data Flow	34
3.1.2.4	Template	35
3.1.3	Collaborative Infrastructure	35
3.2	System Design	36
3.2.1	CLUE	38
3.2.2	Orchestrator	38
3.2.3	C2I	39

3.2.4	D2I	40
3.3	SDN-Based Network Abstraction	41
3.3.1	Physical Network	41
3.3.2	Virtual Network	42
3.3.3	Network Control Plane	43
3.3.4	Data Transfer Mechanism	44
3.4	Priority-Based Global Network Optimization	47
3.5	TCP _{Mario} For Bandwidth-Reservable Networks	50
3.6	Experimental Evaluation	51
3.6.1	Experiment Setup	52
3.6.2	Evaluation of <i>MCF+</i>	54
3.6.3	Evaluation of TCP MARIO	56
3.7	Chapter Summary	58
4	Geo-Distributed, Network-Aware Caching for Data-Intensive Applications	59
4.1	Problem Formulation	60
4.2	System Design	62
4.2.1	Architecture	62
4.2.2	On <i>datapods</i> , <i>dataserver</i> and <i>dataclients</i>	65
4.2.3	Network-Aware Cache Algorithm	67
4.2.4	Computation Sharing	71
4.3	Evaluation	72
4.3.1	Experiment Setup	73
4.3.1.1	System Configuration and Environment	73
4.3.1.2	Workloads	73
4.3.1.3	Baseline Algorithms	74
4.3.2	Synthetic Data Set	74
4.3.2.1	Network-Awareness and Performance Breakdown	74

4.3.2.2	Impact of Network Bandwidth	75
4.3.2.3	Stability of <i>datapods</i> and replicas	77
4.3.3	Real-World Data Set	81
4.3.3.1	Impact of Cache Capacity on Performance	81
4.4	Chapter Summary	83
5	Enable Cost-Aware Scheduling of Applications in a Multi-Cloud Environment	84
5.1	System Design	85
5.1.1	PIVOT Application	85
5.1.2	PIVOT Architecture	86
5.1.2.1	Cross-Cloud Virtual Infrastructure	87
5.1.2.2	Abstraction Layer	89
5.1.2.3	Scheduling Layer	91
5.2	Cost-Aware Scheduling Algorithm	92
5.2.1	Problem Definition	92
5.2.2	Algorithm Design	93
5.2.3	Task Grouping and Data Locality Inference	95
5.2.4	Task Ordering	96
5.2.5	VM Ordering	97
5.2.6	First-Fit Vector Bin Packing	98
5.3	Evaluation	99
5.3.1	Experiment Setup	99
5.3.1.1	Alibaba Cluster Trace and Simulation	99
5.3.1.2	Big Data Applications and Real Deployment	100
5.3.1.3	Baseline	101
5.3.2	Results	101
5.3.2.1	Simulation on Alibaba Cluster Trace	101
5.3.2.2	The Hail Cluster	104

5.3.2.3	CWL Workflows.....	107
5.4	Chapter Summary	108
6	Conclusion and Future Work	109
6.1	Summary of Results	109
6.2	Other Work	111
6.3	Future Work	111
	BIBLIOGRAPHY	114

LIST OF TABLES

3.1	DFD entities	34
3.2	REST API of key components in the network of <i>collaborative infrastructure</i>	45
3.3	Baseline TCP congestion control algorithms	54
5.1	Egress network traffic cost within and between AWS and GCP	99

LIST OF FIGURES

1.1	Excessive egress network traffic and cloud expenses caused by loss of data locality	11
2.1	Comparison of hardware virtualization approaches (Hwang et al., 2013)	18
2.2	Comparison between container-based and traditional virtualization (Zhang et al., 2010)	19
2.3	SDN Architecture (Braun and Menth, 2014)	21
3.1	CLUE Object	33
3.2	RADII architecture	37
3.3	Web GUI of CLUE	38
3.4	Network architecture in a <i>collaborative infrastructure</i>	42
3.5	Examples of input and output of the <i>network planner</i>	46
3.6	Network topology for evaluation	52
3.7	The ECMP algorithm	53
3.8	Comparison of average data transfer throughput between ECMP and MCF+	55
3.9	Throughput variation of 50 data transfers between UFL and UMASS	56
3.10	Variation of overall bandwidth utilization in 30 minutes	57
3.11	Comparison of bandwidth utilization among the selected TCP congestion control algorithms in different network conditions with varying bandwidth, latency and packet loss rate	58
4.1	CACHALOT physical infrastructure	63
4.2	Interaction among the client, CA and CM in CACHALOT	64
4.3	CACHALOT architecture. The bottom layer is the physical infrastructure in which CAs are interconnected via WAN and co-located with clients. The CM coordinates cache operations among CAs. The upper layers are <i>datapod</i> the topology of different data objects. CAs with a crown are the <i>dataservers</i> and <i>dataclients</i> fetch replicas from the <i>dataserver</i> of <i>datapods</i> . Note that a <i>dataclient</i> of one <i>datapod</i> may migrate to another <i>datapod</i> (shown in the middle layer) due to changes in replica distributions and network conditions	66
4.4	CACHALOT cache algorithm performance breakdown	76
4.5	Performance with varying bandwidth	77

4.6	Distribution of <i>datapod</i> number and lifetime	78
4.7	Performance with varying sizes	79
4.8	Performance with varying popularity distribution	80
4.9	Characteristics of OpenCloud and ExoGENI trace data set	81
4.10	Performance with varying capacity	82
5.1	An example of PIVOT application	86
5.2	PIVOT architecture	87
5.3	Cloud regions in AWS and GCP used for evaluating PIVOT	100
5.4	Comparison of egress cost, host subscription cost, average number of VM instances used and application runtime. cost-aware saves up to 90.8% and 99.2% of the cost for host subscription and egress traffic.	102
5.5	Average data transfer time per task. cost-aware saves up to 82.8% of data transfer time due to strategic selection of fast network path and avoidance of network congestion.	103
5.6	Variation of cloud expenses with increasing number of running applications. Despite the lowest egress and host subscription cost, cost-aware achieves mildest cost increase as applications scale up.	104
5.7	CDF of worker co-locations/data transferred for Hail clusters/CWL workflows in different scopes, respectively. cost-aware effectively co- locates the workers/tasks for improved data locality and less egress cost.	105
5.8	Footprints of the Hail cluster deployments in the cost-throughput space. Most deployments by cost-aware are clustered in proximity to the optimum, while those by the baselines mostly distributed at the far end.	106
5.9	Average egress cost incurred by running the CWL workflows. cost-aware saves up to \$2,092 (92.2%) in total for 30 workflow runs	107

LIST OF ABBREVIATIONS

ACK	Acknowledgement
ACL	Access Control List
AES	Advanced Encryption Standard
AIMD	Additive-Increase/Multiplicative-Decrease
API	Application Program Interface
AZ	Availability Zone
BDP	Bandwidth-Delay Product
BLOB	Binary Large Object
BF	Best Fit Algorithm
CDF	Cumulative Distributed Function
CI	Cyberinfrastructure
CPU	Central Processing Unit
CRUD	Create, Read, Update, Delete
CWL	Common Workflow Language
DAG	Directed Acyclic Graph
DFD	Data-Flow Diagram
ECMP	Equal-Cost Multi-Path
FF	First Fit Algorithm
FFD	First Fit Decreasing Algorithm
GD	Greedy Dual Size Frequency Algorithm
HPC	High-Performance Computing
HTB	Hierarchical Token Bucket
HTTP	Hyper Text Transfer Protocol
IaaS	Infrastructure-as-a-Service
I/O	Input/Output
IoT	Internet of Things
IP	Internet Protocol
IT	Information Technology

KB	Kilobyte
LAN	Local Area Network
LDAP	Lightweight Directory Access Control
LFU	Least Frequently Used
LP	Linear Programming
LRU	Least Recently Used
LRV	Least Relative Value
MAC	Media Access Control
MB	Megabyte
MCF	Multi-Commodity Flow
MD5	Message-Digest Algorithm 5
MDVPP	Multi-Dimensional Vector Bin Packing Problem
MKP	Multiple Knapsack Problem
MPTCP	Multipath TCP
MSS	Maximum Segment Size
NDL	Network Description Language
NFS	Network File System
NP	Nondeterministic Polynomial
OF	OpenFlow
OS	Operating System
OLAP	On-Line Analytical Processing
OVS	Open vSwitch
POSIX	Portable Operating System Interface
QoS	Quality of Service
RAM	Random Access Memory
REST	Representational State Transfer
RTT	Round Trip Time
SDN	Software-Defined Network
SLA	Service-Level Agreement

SOA	Service-Oriented Architecture
SPARC	Scalable Processor Architecture
TCP	Transmission Control Protocol
URL	Uniform Resource Locator
VBP	Vector Bin Packing
VLAN	Virtual Local Area Network
VM	Virtual Machine
VMM	Virtual Machine Monitor
VPC	Virtual Private Cloud
VPN	Virtual Private Networking
WAN	Wide Area Network

CHAPTER 1: Introduction

1.1 Data-Intensive Applications

A data-intensive application performs massive data processing activities upon vast amounts of raw data to gain insights into the data and unearth valuable facts to advance research and business, e.g., genomic sequence workflows, business intelligence analytics, among others. The data activities performed by data-intensive applications involve every phase of data processing, including but not limited to acquisition, warehousing, mining, cleansing, aggregation, integration, analysis, and modeling of data. The data being processed and analyzed is also stored and presented in a multiplicity of forms ranging from binary (e.g., program executables) to multimedia content (e.g., videos). The large scale of data, diversity of data activities, and intricacy of data composition characterize a typical data-intensive application nowadays. More specifically, data-intensive applications run on data sets with 3Vs (Assunção et al., 2015; Shah et al., 2015): volume, velocity, and variety – the volume means the sheer amount of data being processed; the velocity indicates the high rate of newly produced inflow data; the variety represents the various data formats and varying levels of data noise. These three factors have a profound influence on the performance of data-intensive applications and deeply impact the architectural and algorithmic design of data-intensive applications and the platforms they are executed on. Hence, our discussion on data-intensive applications in this dissertation centers on these three key factors of data.

1.1.1 Data Volume

The enormity of data by itself poses unprecedented challenges to data-intensive applications and their underlying infrastructure: Dobre and Xhafa (2014) report that around 2.5 exabytes of raw data are being produced in the world on a daily basis – YouTube receives 48 hours of video uploads per minute and the cumulative data volume uploaded in two months surpasses the total of three major television broadcasters (Fox, 2011); Walmart reportedly collects more than 2.5 petabytes of transaction data every hour from its customers (Sivarajah et al., 2017). The colossal amounts of data can rapidly overwhelm data storage capacity

in any single data center, prohibiting traditional centralized data processing (Corbett et al., 2013; Gupta et al., 2014; Hwang et al., 2007). Furthermore, the data is highly distributed, generated by devices, individuals and organizations from around the world. The ubiquity of data sources and sparse distribution of data necessitate data transport for data analytics, which is hindered by the lack of network bandwidth in WANs – massive data transfers can easily oversubscribe network capacity and stall due to severe network congestion (Pu et al., 2015; Vulimiri et al., 2015a,b). This phenomenon, referred to as Data Deluge, is recognized as a clear trend (Fox, 2011) and calls for highly scalable design of data-intensive applications and the underpinning infrastructure to cope with the ever-growing data volume.

Further, traditional scaling models (Amdahl, 1967; Gustafson, 1988; Sun and Chen, 2010) have inadequacies in modeling data-intensive applications because of their oversight of the huge data volume. Traditional models mostly focus on the speedup achieved by data parallelism but ignoring the overhead induced by scaling, e.g., task dispatching, inter-task communication, among others (Li et al., 2019). This state of affairs is due to the fact that these models are designed for high-performance computing (HPC) systems wherein it is assumed that the high-speed local area network (LAN) has abundant network capacity and transmission of moderate volume of data takes a trivial amount of time as compared to data processing time. This assumption does not hold true for the majority of data-intensive applications since data processing activities are mostly I/O-bound in nature and thereby up to an order of magnitude more time-consuming than their memory-intensive counterpart. Consequently, it is common for data transmission time to dominate the end-to-end execution time of data-intensive applications due to the slowness of frequent data processing activities.

Moreover, the massive data volume also has significant financial implications in the era of cloud computing, since data-intensive applications are migrating from on-premise systems to cloud platforms to utilize computing resources in an on-demand manner (Buyya et al., 2009). As cloud providers monetize data by units being stored, transmitted, and processed in the cloud, cloud expenses skyrocket for data-intensive applications. Hence, a plethora of researches (Wu et al., 2013, 2015; Hsieh et al., 2017; Chung et al., 2018) concentrate on improving cost effectiveness of data-intensive applications in the cloud by saving superfluous network traffic and data dumps in store.

1.1.2 Data Velocity

Despite the gigantic data volume, the high rate of data influx is another major challenge faced by data-intensive applications. For instance, Walmart processes over a million transactions per hour (Cukier, 2010); Square Kilometer Array Telescope generates 100 terabits per second; exascale simulation produces result dumps at the rate of terabytes per second (Fox, 2011). The torrent of data continuously stresses the applications and infrastructure underneath. Therefore, it is imperative for data infrastructures to guarantee high reliability and fault tolerance for data-intensive applications.

With data constantly flooding in, some applications impose more stringent time constraints on the processing of data to pace up with the rapid data arrival and capture momentary changes in data. For instance, automatic trading systems continuously monitor stock prices and need to react to instant price spikes in seconds; retail stores need to analyze data collected from users' mobile devices (e.g., location) to generate personalized promotion offers in minutes (Gandomi and Haider, 2015); online robotic dialog systems (e.g., Slackbot¹) are required to comprehend and respond to user interactions promptly. Assunção et al. (2015) categorize data-intensive applications into four types based on the strictness of time constraints, which are batch, near-time, real-time, and streams. Particularly, data velocity has the most salient impact upon real-time, e.g., data analytics queries (Rabkin et al., 2014; Pu et al., 2015; Vulimiri et al., 2015a), and streams, e.g., data streaming applications (Xu et al., 2014; Peng et al., 2015; Caneill et al., 2016). Given more rigorous time constraints, low latency is of greater importance to these time-critical applications than high throughput due to costly penalties for delays in data processing.

Latency arises mainly due to shortage in processing ability and data movements – data processing tasks may contend for limited computing cycles and forcibly queue up to wait for next available ones, causing queuing delay; communication and data sharing among the tasks initiate data movements and unavoidably incur data propagation delay. To mitigate the queuing delay, it is requisite to invest more in computing resources to resolve the bottleneck in resource provisioning and guarantee abundant computing cycles. More importantly, applications should be highly scalable by design, able to scale out and exploit additional resources. Data propagation delay is a product of round trip time (RTT) between tasks and network congestion. The RTT is largely determined by the speed of light and therefore it is difficult to overcome, while

¹An Introduction to Slackbot: <https://slack.com/help/articles/202026038-An-introduction-to-Slackbot>

network congestion forces packet loss and retransmission, bogging down data transfers over the network. To reduce data propagation delay, computer networking studies have proposed a series of network optimization approaches to circumventing or alleviating network congestion (Al-Fares et al., 2010; Hong et al., 2013; Alizadeh et al., 2014; Cai et al., 2016; Cardwell et al., 2016; Langley et al., 2017). On other alternative approach that has gained much attention in the community for its effectiveness in enabling low-latency data analytics is to retain data locality and shorten the traveling distance of data movements (Hindman et al., 2011; Zaharia et al., 2012; Pu et al., 2015; Caneill et al., 2016).

1.1.3 Data Variety

Data variety refers to the heterogeneity found in data forms, provenance, crudity and quality of data-intensive applications (Sivarajah et al., 2017). Data to be processed ranges from unstructured free text to annotated video snippets; some data sets are provided in raw formats, while others are excerpts of original data. To make things worse there are also different levels of noise across various data sets. This heterogeneity adds to the difficulty for data-intensive applications to comprehend and make sense of data. Further, the data variety forces applications to use a diversity of data processing frameworks among which interoperability is absent and data formats are incompatible, therefore further worsening the heterogeneity challenge. Hindman et al. (2011) reveal that lack of interoperability among data processing frameworks prevents applications from scaling out and harms data locality. From the perspective of system administration, maintaining a large number of frameworks is laborious and error-prone, costing a non-trivial, increase in IT investment.

Additionally, in this dissertation, we consider data variety to encompass the highly variable patterns of data generation and processing that results in load spikes that affect performance of data-intensive applications. Particularly, for most user-driven applications (e.g., mobile applications), data load is unpredictable because of spontaneous user behaviors. This variety induces a radical implication on system elasticity and scalability. Without an elastic infrastructure, traditional approaches address workload bursts by overprovisioning, which wastes resources and energy. Today, enabled by cloud computing, data-intensive applications can use computing resources as a utility (Buyya et al., 2009) allowing data processing systems to scale dynamically to accommodate varying resource demands of applications. Meanwhile, data-parallel models (Isard et al., 2007; Dean and Ghemawat, 2008) immensely improve the application scalability across commodity servers. However, the system elasticity and scalability are currently limited by the capacity of individual cloud providers Buyya et al. (2010); Varghese and Buyya (2018). Vendor lock-ins and business barriers among

providers prevent applications from scaling across cloud platforms to gain access to and make use of more resources across cloud platforms. As cloud competitors inherently lack the initiative to promote inter-cloud application scaling, broker services that orchestrate applications across clouds need to be developed to achieve higher level of elasticity and scalability (Grozev and Buyya, 2014).

1.2 Geo-Distributed Computing

A massive scale of data is being generated at very high rates all over the world every day, e.g., emails, ad clicks, online purchases, social media posts, temperature readings, and many others. Particularly, with the development of mobile technologies and the emergence of Internet of Things (IoT) (Li et al., 2015), intelligent portable devices, e.g., smart phones, smart home devices and sensors, are widely equipped and increasing the already formidable data volume worldwide. Major IT service providers, e.g., Google (Calder et al., 2013; Jain et al., 2013), Microsoft (Hong et al., 2013), Amazon² and AT&T (Hung et al., 2015), deploy tens to hundreds of global data centers to acquire, store and analyze geo-distributed data sets. Example analyses include querying user clicks to aid advertisement recommendation (Corbett et al., 2013), querying server logs to monitor system health, querying network logs to detect malicious attacks, etc. These data analytics applications are data-intensive in nature, producing a large scale of intermediate and output data from distributed geo-locations.

Traditional data analytics frameworks still adopt the centralized data processing model (Pietzuch et al., 2006; Corbett et al., 2013; Gupta et al., 2014; Kraska et al., 2013). With this model, geo-distributed data sets are transferred to a single central data center over WANs for aggregation and analysis. The centralized model has several critical disadvantages in a geo-distributed setting. First, centralized data processing heavily stresses the central data center in computing, storage and power provisioning, and greatly limits scalability of applications to the capacity of the data center, which is nontrivial to scale on demand to handle load bursts. Second, backhauling data over WANs is inefficient from the perspective of resource utilization and application performance. Transferring large amounts of data between geo-distributed data centers incurs substantial network traffic and wastes valuable WAN bandwidth. On the other hand, application performance will be impeded by remote data transfers over scarce network resources prolonging the application execution time. In addition to the bandwidth shortage, the huge volume of data may overflow buffers of top-of-the-rack

²Global Infrastructure – Amazon Web Services: <https://aws.amazon.com/about-aws/global-infrastructure/>

switches in data centers, causing a known network problem referred to as TCP incast (Chen et al., 2012b) and further degrading the performance of TCP data transfers. Third, centralized data processing is susceptible to single point of failure upon data center outage and affects reliability of applications. Last but not least, a centralizing approach is often prohibited by data privacy and sovereignty laws, which limit data movements within only authorized domains or locations.

Geo-distributed computing has emerged as the de-facto model for performing geo-distributed data analytics since it addresses the limitations of the centralized model. Following, we elaborate on the benefits of geo-distributed computing for data-intensive applications.

1.2.1 High Scalability

Geo-distributed computing model allows data-intensive applications to scale out across multiple geo-distributed data centers rather than be confined to a single one. The ability to scale out is essential to data-intensive applications since their workloads are bursty, and unexpected workload spikes may exceed the capacity of a well resourced data center. For instance, in our experiments we have encountered such resource shortage in Google’s `us-east4` data center at North Virginia. More specifically, a RNA sequence alignment workflow we use to drive experiments was frequently unable to scale out to other data centers in Google Cloud due to insufficient resources. Upon this situation, pending jobs would block long periods of time impacting significantly the overall execution time of the workflow. Data centers typically overprovision resources to cope with workload bursts (Grozev and Buyya, 2014), leaving data centers underutilized for the majority of time and incurring wasteful power consumption. In the geo-distributed model, applications gain access to a large pool of resources from multiple geo-distributed data centers. If resources in a data center are depleted, applications can scale out to other data centers to continue making progress instead of halting for long periods of time. Hence, geo-distributed computing enhances the scalability of applications and lowers the risk of resource shortage.

1.2.2 Low Latency

Geo-distributed computing offer low latency capabilities not possible with other computing models in similar environments. An outstanding characteristic of geo-distributed computing is the low latency it achieves for data-intensive applications running on top of geo-distributed data sources and data sets. By spreading data processing jobs across the geo-distributed data centers, applications gain proximity to the data

they need and reduce or even eliminate the latency they would have experienced otherwise. As discussed in the prior section, low latency is crucial for time-critical, data-intensive applications such as real-time data analytics queries and data streaming applications. A plethora of studies (Wu and Madhyastha, 2013; Pu et al., 2015; Hung et al., 2015; Wu et al., 2013) have shown that increasing density of geo-distributed data centers is conducive to lowering network latency for geo-distributed applications. In practice, using multiple cloud platforms is a viable way to improve geospatial coverage of data centers since cloud providers collectively have widespread data center deployments globally. It is worth mentioning that certain data operations such as aggregation must always be done in a centralized manner. However, the geo-distributed model can support data cleansing, selection and partial data aggregation at the edge to reduce the amount of data to be backhauled for final centralized processing, therefore greatly shortening the time for data transmission over WANs (Rabkin et al., 2014). This has a huge implication on user-perceived latency of applications since time for data transfers often dominates the end-to-end runtime of applications.

1.2.3 Strong Resilience

Relying on a single data center exposes data-intensive applications to potential breakdown caused by data center failures, which are not uncommon as observed in recent years. Major cloud providers, including Amazon Web Services (AWS)³, Microsoft Azure⁴ and Google Cloud Platform (GCP)⁵, have all experienced disruptive data center outages in the past few years. A post-mortem analysis performed by Amazon⁶ even advised their clients to distribute their applications across multiple data centers for fault tolerance. Although cloud providers today deploy multiple independent clusters at every single region, referred to as availability zones (AZs), to lower the risk of total data center breakdown, site-wide failures are still possible due to unexpected disasters. The geo-distributed computing model uses multiple geo-distributed data centers as insurance against potential breakdown of one or multiple data centers, greatly lowering the risk of application breakdown and ensuring strong resilience of applications.

³Summary of the AWS Service Event in the US East Region: <http://aws.amazon.com/message/67457>

⁴Windows Azure Service Disruption Update: <http://blogs.msdn.com/b/windowsazure/archive/2012/03/01/windows-azure-service-disruption-update.aspx>

⁵Post-mortem for February 24th, 2010 outage: https://groups.google.com/group/google-appengine/browse_thread/thread/a7640a2743922dcf?pli=1

⁶Summary of the Amazon EC2 and Amazon RDS Service Disruption: <http://aws.amazon.com/message/65648>

1.2.4 Regulatory Compliance

Valuable data sets are mostly sensitive and proprietary. Regulatory constraints are commonly imposed on these data sets to protect the intellectual properties and data sovereignty (Mohan et al., 2012; Vulimiri et al., 2015b). These regulatory policies typically prevent protected data sets from being physically moved around at will to ensure data security. For instance, European Union forbids transfer of personal data to non-member countries without an adequate level of protection measures (Varghese and Buyya, 2018); most cloud providers are compliant to Healthcare Insurance Portability and Accountability⁷, prohibiting sensitive patient health information from being disclosed without the patient’s consent and knowledge. These legislative and regulatory constraints prevent centralized data aggregation and processing for data-intensive applications running on top of the most sensitive and proprietary data. Instead of moving sensitive data to a central data center, geo-distributed computing has the flexibility to distribute data processing tasks to data centers with proper security clearance and authorization to perform in-place data analytics without violating regulatory constraints. This capability ensures regulatory compliance of data-intensive applications.

1.2.5 Cost Effectiveness

Despite performance degradation, transferring colossal amounts of data sets over WANs is prohibitively expensive due to the huge consumption of WAN bandwidth it incurs, which is scarce and charged for usage at a high unit price. Most cloud providers charge for egress network traffic produced by applications, which is the outbound network traffic from a geo-location such as a data center in the cloud; the rate is increased if the traffic goes outside the cloud platform to the Internet. For instance, GCP charges \$0.01 per gigabyte for network traffic traveling among its cloud regions in United States and Canada but penalizes egress Internet traffic by a rate of \$0.12 per gigabyte – a 12x price increase. Other providers such as AWS and Microsoft Azure have a similar pricing model for egress network traffic. Reserving dedicated network links between geo-locations can reduce egress network traffic cost, since cloud providers usually charge a one-time fee for setting up network circuits independently of the network traffic. However, the initial investment for such dedicated network links is typically too high to be amortized for applications at small to medium scale. Hence, a large number of research efforts focus on reducing the amount of data transferred over WANs for

⁷Health Insurance Portability and Accountability Act of 1996 (HIPAA): <https://www.cdc.gov/phlp/publications/topic/hipaa.html>

data-intensive applications in geo-distributed environments to save the monetary cost associated with egress network traffic (Wu et al., 2013, 2015; Pu et al., 2015; Kloudas et al., 2015; Hsieh et al., 2017). With the geo-distributed model, data pre-processing and partial aggregation are performed at the edge, significantly reducing the amount of data being transferred among geo-distributed data centers and, thus, saving the egress network traffic cost (Rabkin et al., 2014). Additionally, as data-intensive applications are being migrated to the cloud in nowadays, geo-distributed computing can take advantage of the price discrepancies among cloud providers to minimize the cloud expenses for resource subscription (Wu et al., 2013). For instance, AWS offers resources with variable prices, i.e., Spot Instances⁸, which change based on the supply and demand in the market can be exploited for cost saving.

1.3 Challenges for Data-Intensive Applications in Geo-Distributed Environments

In the prior sections, we elaborate on the characteristics of data-intensive applications and advantages of a geo-distributed computing model for running these applications upon large-scale, geo-distributed data sets. In this section we focus on three key challenges we have identified which hinder running data-intensive applications in geo-distributed environments.

1.3.1 High Degree of Heterogeneity

Geo-distributed environments are intrinsically heterogeneous. They typically consist of loosely-coupled, geo-distributed infrastructures, which are designed, built and maintained independently from each other. At low level, geo-distributed data centers employ heterogeneous hardware and software from central processing units (CPU) to operating systems (OS). Servers in different racks even use disparate CPU platforms; some nodes are equipped with accelerators such as graphic processing units (GPU), field-programmable gate array (FPGA) and many-core processors (e.g., Intel Xeon Phi) – a variety of OS, including Microsoft Windows Server, Solaris, BSD family, and miscellaneous Linux distributions, are running on different servers. Standardization of Application Programming Interfaces (APIs) is limited thus impeding interoperability across data intensive applications running on diver high-level frameworks. Furthermore, network heterogeneity is even more severe in geo-distributed environments as WANs are made of heterogeneous subnets backed by various network providers – despite the high level of heterogeneity in network devices (e.g., assorted models

⁸Amazon EC2 Spot Instances:<https://aws.amazon.com/ec2/spot>

of network switches and routers), there are plenty of discrepancies in network architecture and management among providers. Consequently, heterogeneous WANs are susceptible to connectivity issues and network congestion, causing underwhelming and unpredictable network performance. As summarized by Varghese and Buyya (2018), heterogeneity exists in nine layers of a geo-distributed system from bottom up, including network, storage, servers, virtualization, OS, middleware, runtime, data, and application.

Particularly, as geo-distributed computing is often realized on top of multiple cloud platforms in practice (Stefanov and Shi, 2013; Wu et al., 2013; Jonathan et al., 2016), there is also heterogeneity in cloud services and pricing models. Different cloud providers expose heterogeneous APIs for subscribing, managing and releasing miscellaneous resources, impeding data-intensive applications from scaling across clouds to leverage featured services offered by specific vendors and alleviate platform-wide risks (e.g., political or legislative restrictions). Furthermore, cloud providers are disincentivized to improve interoperability among cloud platforms, relying on lock-in strategists to discourage clients from migrating to the competitors. This further exacerbates the heterogeneity in the cloud. Although plenty of efforts have been made to bridge the gap between cloud platforms (Grozev and Buyya, 2014), they are still preliminary, and deeper integration among cloud platforms is needed to mitigate the heterogeneity.

1.3.2 Loss of Data Locality

As discussed in prior sections, geo-distributed computing has the potential to improve data locality for data-intensive applications by co-locating computation and data. Nevertheless, it is difficult to directly adapt well-known data processing frameworks, e.g., Hadoop (Shvachko et al., 2010), Spark (Zaharia et al., 2010b), and others (Isard et al., 2007; Malewicz et al., 2010), to geo-distributed environments. Although these frameworks are originally designed to optimize for data locality, they are oblivious to the heterogeneity commonly found in compute infrastructure and therefore are only suitable for cluster computing within single data centers. Consequently, they may worsen data locality for geo-distributed environments thus affecting the performance of data-intensive applications. Figure 1.1 shows the egress network traffic and cloud expenses incurred when deploying a Mesos (Hindman et al., 2011) instance into a multi-cloud, geo-distributed environment. Without taking into consideration the geographic distribution of resources and egress network traffic cost, the default scheduler of Mesos spreads out compute tasks across the geo-distributed resources incurring a prohibitive amount of costly egress network traffic. In fact, our experiments show that almost 90% network traffic travels across cloud regions and platforms, inducing 98.5% of total cloud expenses. As

expected, long-distance data transfers slow down data-intensive applications given the large volume of data to be transferred over the WAN. We will further demonstrate the negative impact on application performance caused by loss of data locality in the experimental evaluation introduced in the following chapters.

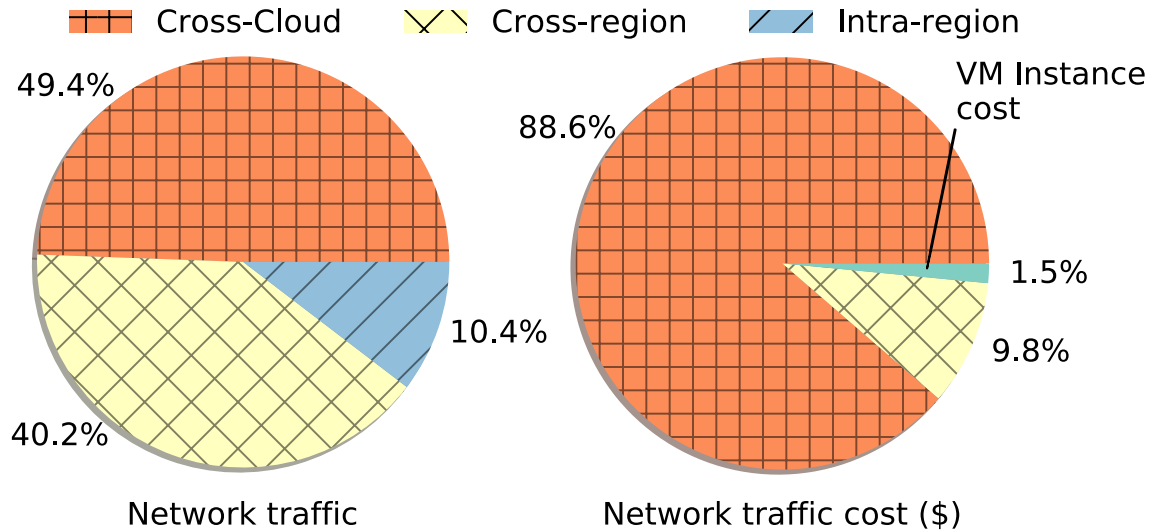


Figure 1.1: Excessive egress network traffic and cloud expenses caused by loss of data locality

There are two lines of work in improving data locality in geo-distributed environments. One line of work focuses on task scheduling optimization (Vulimiri et al., 2015a; Pu et al., 2015; Viswanathan et al., 2016; Jonathan et al., 2016). Another line of work focuses on data caching and replication (Wu et al., 2013; Corbett et al., 2013; Stefanov and Shi, 2013; Gupta et al., 2014) to improve data locality in geo-distributed environments. These approaches have a piecemeal nature, while comprehensive and generalizable solutions are still absent. We believe that proper abstraction of compute resources is fundamental to preserving data locality in geo-distributed environments since it can standardize and enrich information about the environments for high-level scheduling processes.

1.3.3 Inefficiency in Data Sharing

Large data transmission over the network is inevitable for data-intensive applications even with optimal data locality, since data processing tasks generate and share a large amount of intermediate output with each other. For instance, for MapReduce (Dean and Ghemawat, 2004) applications, the `shuffle` phase shuffles the output data of `mappers` among `reducers`, resulting in intensive data sharing over the network. Further, machine learning models commonly create huge parameter matrices, which are frequently updated

by distributed processes (Hsieh et al., 2017). In addition, tasks of genomic alignment workflows generate pre-aligned intermediate results consumed by its next steps running on distributed nodes. Hence, it is crucial to ensure the efficiency of data sharing to satisfy quality-of-service (QoS) requirements imposed on data-intensive applications.

WAN performance is key to achieving high efficiency of data sharing in geo-distributed environments since the majority of data transmissions rely on WANs. However, it is common for geo-distributed applications to encounter performance issues in WANs. First, most WANs suffer from constrained bandwidth and high latency, therefore increasing the likelihood to network congestion. Second, traditional WAN architecture has design limitations that hinder network optimization, leading to several performance challenges in the operation and use of WANs (McKeown et al., 2008; Braun and Menth, 2014). Finally, infrastructure for supporting high-speed WAN is lacking. More specifically, advanced WAN solutions such as Multi-Protocol Label Switching Traffic Engineering (MPLS-TE) (Meyer and Vasseur, 2010) are proprietary and expensive to adopt; and, data transfer infrastructures such as SDX (Gupta et al., 2015), are only available in highly specialized settings. Commonly, data-intensive applications resort to the Internet for remote data transfers, which is extremely inefficient. It took us over two weeks to transfer 5 terabytes of hydrology data sets between two institutions in the United States. On the other hand, data processing frameworks lack WAN awareness to circumvent performance bottlenecks, since limited bandwidth, high latency and high probability of packet loss are unique in WANs and rarely considered in the design of data center solutions. Enhancing network awareness in data-intensive applications can facilitate the efficiency of data sharing in geo-distributed environments.

1.4 Thesis Statement

In the recognition of these critical challenges, we seek to address each of these challenges systematically in this dissertation. This leads to the following thesis statement:

Data-intensive applications face high levels of heterogeneity, loss of data locality and inefficiency in data sharing in geo-distributed environments, which create significant detriments to application efficiency. To improve the efficiency, these challenges ought to be tackled via resource abstraction, preservation of data locality and network optimization.

1.5 Contributions

To support the aforementioned thesis, we summarize the contributions made in this dissertation in the following three aspects.

1.5.1 Resource Abstraction

As explained in Section 1.3.1, heterogeneity exists in many aspects of geo-distributed computing from low-level infrastructure to high-level representation of data-intensive applications. Heterogeneity is recognized as the major culprit impeding data-intensive applications from running smoothly in geo-distributed environments. Hence, it is imperative to first address all issues related to heterogeneity in order to further improve the efficiency of data-intensive applications.

In this dissertation, we develop resource abstraction mechanisms across different levels of geo-distributed computing systems to mitigate the aforementioned heterogeneity. Specifically, in Chapter 3, we introduce a high-level abstraction based on the data-flow diagram (DFD) model to simplify the description of data-centric scientific collaborations and bridge the gap between computing and data in scientific collaborative applications. We develop a virtual network layer using software-defined networking (SDN) to abstract the highly heterogeneous WANs connecting geo-distributed, participating research institutions, addressing not only the heterogeneity found in the WANs but also opening the door to complex network optimization. In order to generalize caching of computation, in Chapter 4, we characterize jobs of data-intensive applications based on their key properties. Further, we create an abstraction layer to standardize access and use of cache spaces contributed by a multiplicity of geo-distributed servers. In Chapter 5, we achieve cloud agnosticism across geo-distributed cloud regions by creating an abstraction of compute resources provisioned by multiple cloud providers, allowing seamless scaling and migration of data-intensive applications across geo-distributed cloud regions and multiple clouds. These pieces of work lay the groundwork for other contributions, which would otherwise be intractable in geo-distributed environments.

1.5.2 Preservation of Data Locality

Data locality is central to data-intensive applications, especially in geo-distributed environments, because of its enormous impact on application performance and associated monetary cost as described in Section 1.3.2.

In this dissertation, we are committed to preserving data locality for data-intensive applications in geo-distributed environments.

In Chapter 3, we develop a mechanism that allows users to create data stores at geo-locations in proximity to their applications to ensure data locality. In Chapter 4, we develop a network-aware cache network spanning multiple geo-locations to cache and reuse intermediate and output data of repeatedly executed data-intensive jobs thus retaining data locality for applications. As part of this work, we devise an intelligent, network-aware cache algorithm that dynamically caches and replicates data across geo-distributed cache servers to achieve global optimum of data locality and shorten job completion time. Additionally, in Chapter 5, we introduce a novel cost-aware scheduling algorithm that factors in cloud expenses incurred by resource subscription and egress network traffic and co-locates application tasks intelligently across cloud platforms, therefore preserving data locality and minimizing cloud expenditures. These solutions can jointly preserve data locality for data-intensive applications in geo-distributed environments and improve their efficiency.

1.5.3 Network Optimization

We also contribute to the field of network optimization to improve data transfers over WANs. The Internet is unreliable for supporting critical applications, while specialized high-speed WANs are rarely available to the public sector. Furthermore, most applications are oblivious to the unique characteristics of WANs and geo-distributed environments, thus unable to fully utilize the network resources offered by WANs. Last, heterogeneity in WANs aggravates the inefficiency in WANs.

Hence, our contribution in network optimization is three-fold. First, we propose feasible approaches to establish efficient virtual WANs over commodity resources and public cloud services as introduced in Chapter 3 and 5, respectively; these approaches create the engineering fabric for various advanced network optimizations on top. Second, we enhance the network awareness of applications to optimize WAN utilization. Specifically, in Chapter 3, we develop an SDN-based mechanism that maximizes bandwidth utilization in the WAN with respect to priorities of applications. We also pioneered TCP MARIO, a sender-side TCP congestion control variant, which takes explicit bandwidth allocation to TCP connections to maximize throughput of data transfers. In Chapter 4, we introduce network awareness into the caching process of geo-distributed computing, which makes dynamic decisions of data caching and replication in response to ever-changing network conditions. In Chapter 5, we consider the significance of egress cost in budgeting data-intensive applications and enhance the bin-packing task scheduling algorithm by introducing cost awareness with

respect to resource subscription and egress network traffic in the cloud. Last, as explained in Section 1.5.1, we rely on network abstraction to mitigate the heterogeneity in WANs and therefore clear the path for complex network optimization.

1.6 Organization

The rest of this dissertation is organized as follows. In Chapter 2, we provide the background and prior works related to improving efficiency of data-intensive applications and geo-distributed computing. In Chapter 3, we introduce our work on RADII in which we tackle the challenges in describing data-centric scientific collaboration and performing efficient remote data transfers through resource abstraction and network optimization. Next, we present CACHALOT in Chapter 4, in which we develop a network-aware cache network and algorithm for geo-distributed, data-intensive applications to improve applications' data locality. In Chapter 5, we elaborate on our cloud-agnostic solution named PIVOT for geo-distributed, data-intensive applications in the cloud. Lastly, we conclude this dissertation and propose potential future works in Chapter 6.

CHAPTER 2: Background and Prior Work

In this chapter, we present the background for this dissertation by surveying the related prior work.

The organization of this chapter is as follows: we first introduce resource abstraction techniques in Section 2.1; then, we focus on approaches to achieving data locality in distributed systems in Section 2.2; lastly, we present the related works on TCP optimization in WANs in Section 2.3.

2.1 Virtualization-Based Resource Abstraction

Virtualization is a technique of creating software-based virtual resources on the basis of their physical counterparts such as computing, storage, and networking devices and platforms. It is commonly adopted in distributed systems and cloud computing to achieve resource abstraction and enable dynamic, scalable distributed applications.

The idea of virtualization originates as a time-sharing solution on mainframe computers, which creates logically isolated environments for users and applications to share the resources with minimal interference with each other. Today, virtualization has been extensively applied to abstracting other types of resources such as memory, storage, network, among others. Particularly, network virtualization abstracts network devices using software components and creates virtual network spaces, creating opportunities for more dynamic and sophisticated network optimization and enhancement. Recently, lightweight virtualization approaches, e.g., containerization, have emerged to enhance the granularity and agility of distributed application deployment. These modern virtualization technologies lay the groundwork for national cyberinfrastructures (CIs) and cloud computing platforms of today, where geo-distributed, data-intensive applications are primarily deployed and executed on a mass scale.

2.1.1 Hardware Virtualization

Hardware virtualization abstracts a portion or the entirety of physical resources on a computer to multiplex the resources among multiple users and applications. It is initiated on mainframe computers to

enable time-sharing of resources among users and applications. Modern hardware virtualization extends the technique to commodity computers using hardware emulation and software simulation. Specifically, full virtualization techniques rely on 100% binary translation to emulate privileged instruction sets embedded in the hardware and create isolated virtual environments referred to as virtual machines (VMs) in parallel on a single machine, which allows the OS and applications to run on top without any modification as if on a bare-metal machine. Magnusson et al. (1998) first demonstrate the full virtualization of unmodified Linux and Solaris on the SPARC v8 architecture. Devine et al. (2002) propose a virtual machine monitor (VMM) that enables full virtualization on the x86 architecture. Bellard (2005) develops a portable machine emulator that supports full virtualization across multiple CPU architectures. Kivity et al. (2007) introduce a Linux kernel-based VMM named `kvm` specialized for full virtualization on Linux platforms. These works built the foundation for hardware virtualization of today. However, full virtualization is recognized heavyweight by Barham et al. (2003) due to the substantial overhead caused by the full binary translation – certain operations and tasks are significantly time-consuming when performed in the virtual domain as compared to the physical domain. To alleviate the overhead, hardware enhancement, e.g., Intel VT-x (Uhlir et al., 2005), has been developed to provide extra support for circumventing the inefficient binary translation. As an alternative to full virtualization, Whitaker et al. (2002) and Barham et al. (2003) propose the concept of *para-virtualization*, which presents hardware instructions in the form of a software interface and defers the implementation of resource abstraction to the OS running on top. This approach mitigates the performance overhead and can be ported to commodity hardware without any specialized assistance. However, it requires OS modifications to make the OS aware of the virtualized environment and interface to properly execute instructions. Hwang et al. (2013) compare the aforementioned virtualization approaches as depicted in Figure 2.1.

Hardware virtualization underpins most well-known Infrastructure-as-a-Service (IaaS) providers including mainstream cloud platforms and academic cloud testbeds, providing the fundamental low-level resource abstraction for modern distributed computing, especially cloud computing. The infrastructure we have built in this dissertation is laid on hardware-virtualized resources provisioned from distributed geo-locations.

2.1.2 OS-Level Virtualization

The OS-level virtualization, idiomatically referred to as containerization, further abstracts the essential functionalities provided by the OS kernel, i.e., system calls, to allow sharing of an OS kernel among user-space

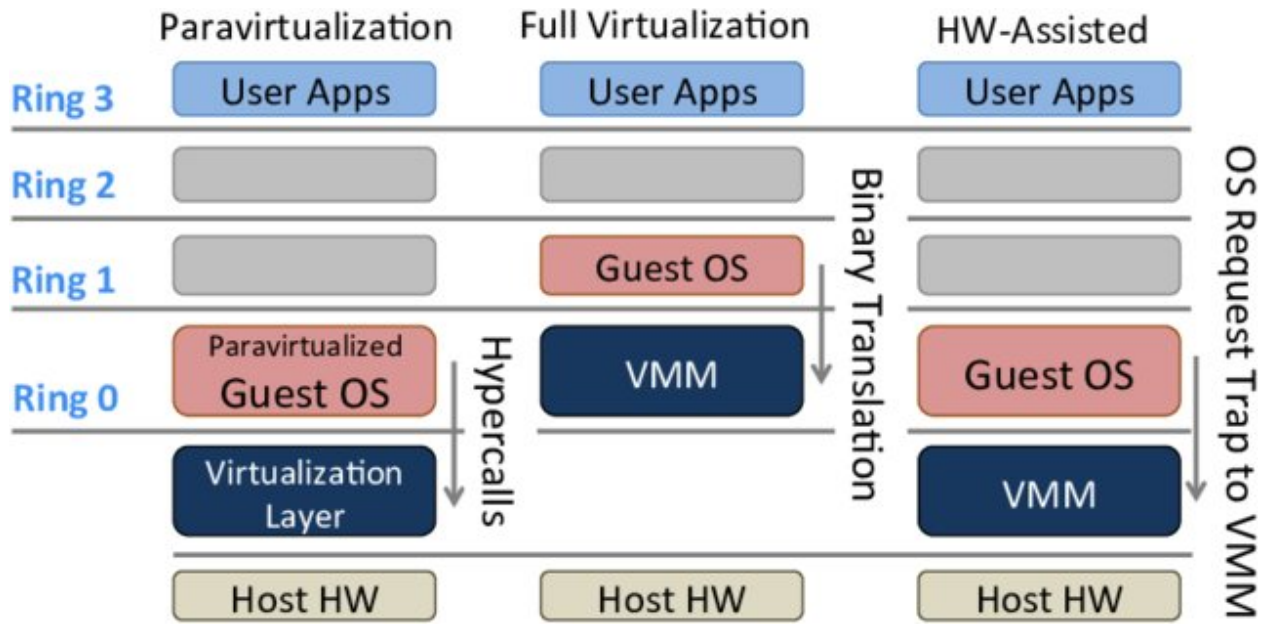


Figure 2.1: Comparison of hardware virtualization approaches (Hwang et al., 2013)

applications running on top. Specifically, it encapsulates applications in a compact and fine-grained fashion, by packaging only application-specific assets, e.g., executables, runtime, and libraries, into isolated containers but sharing other OS resources, e.g., kernels, among applications. In contrast with hardware virtualization, the OS-level virtualization eliminates the overhead for hardware emulation and software simulation; meanwhile, it also avoids resource waste caused by redundant loading of duplicate OS resources as commonly practiced in the one-task-per-instance model (Mao et al., 2010; Van den Bossche et al., 2010; Bittencourt and Madeira, 2011; Mao and Humphrey, 2011), achieving agile application scaling and fast failover in distributed systems.

There have been many academic and industrial efforts devoted to realizing and improving OS-level virtualization. Pahl et al. (2017) conduct a comprehensive taxonomical review of state-of-the-art techniques and methods of containerization and its orchestration. Notably, the introduction of `cgroups`¹ enables the isolation of resource usage by processes in Linux, laying the foundation for modern container techniques. Docker² streamlines the use of containers and boosts the development of microservices architecture (Dragoni et al., 2017). `containerd`³ is an active open-source project in which industry standards for container runtime are being defined and adopted. These containerization techniques facilitate resource abstraction

¹`cgroups`: <http://man7.org/linux/man-pages/man7/cgroups.7.html>

²`Docker`: <https://www.docker.com/>

³`containerd`: <https://containerd.io>

for data-intensive applications. Hindman et al. (2011) propose to encapsulate data processing tasks of data-intensive applications using containers in order to remove the coupling between computing clusters and application frameworks and allow multiple application frameworks to co-exist on a single cluster to improve resource utilization.

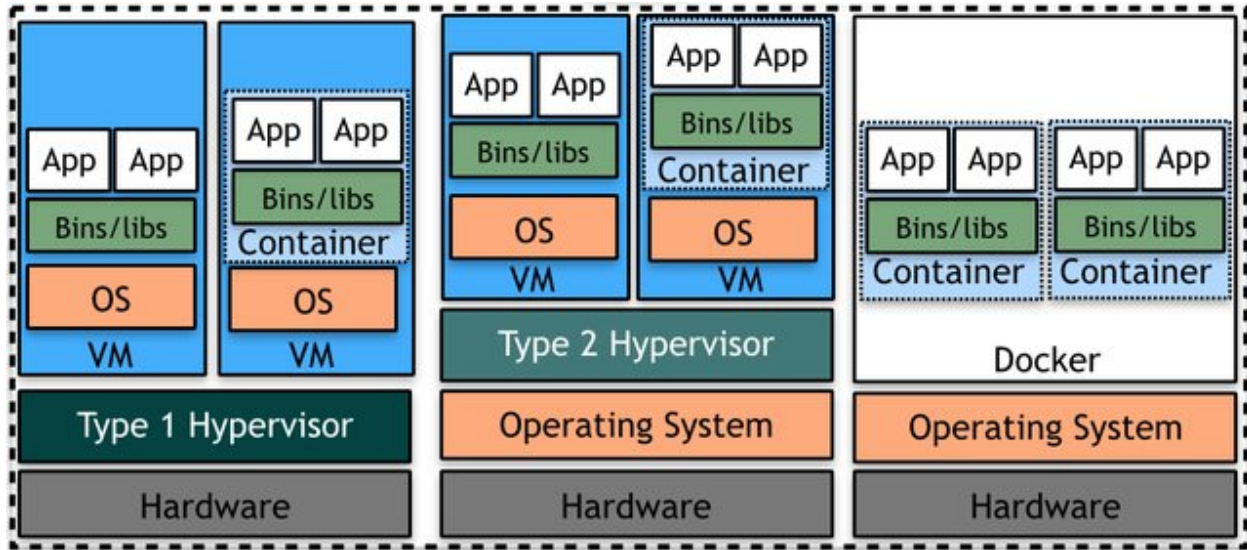


Figure 2.2: Comparison between container-based and traditional virtualization (Zhang et al., 2010)

Zhang et al. (2010) conduct an in-depth survey on virtualization technologies adopted in the cloud and compare different virtualization strategies as illustrated in Figure 2.2. With traditional hardware virtualization, applications are running in units of VMs loaded with full-featured OS. Consequently, it takes extra time to load OS assets before starting applications, slowing down application start-up. In the meantime, application migration and failover are time- and resource-consuming, since the VM hosting the migrating application needs to be transferred from one machine to another in its entirety. In comparison, the container-based solutions enabled by services, such as Amazon Elastic Container Service⁴ and Google Kubernetes Engine⁵, are more lightweight since they deploy and migrate only application assets, which are typically smaller than an entire OS in size. Further, by combining OS-level and hardware virtualization as shown in Figure 2.2, users can have more freedom in customizing resource management for applications running in the cloud – they can develop custom algorithms for scheduling VMs and containers rather than delegate the resource management to cloud providers, and therefore reap more benefits granted by resource abstraction. We

⁴Amazon Elastic Container Service: <https://aws.amazon.com/ecs>

⁵Google Kubernetes Engine: <https://cloud.google.com/kubernetes-engine/>

leverage the combination of OS-level and hardware virtualization to enable the cost-aware cloud agnosticism as introduced in Chapter 5.

2.1.3 Network Virtualization

Network virtualization abstracts network resources and functionalities of physical networks by creating virtual networks connecting distributed, virtualized resources and enabling dynamic software-defined networking (SDN). A virtual network is established over segregated physical network partitions, providing connectivity for distributed applications and presenting a simple logical network topology. It can be created using techniques including virtual local area network (VLAN) (McPherson and Dykes, 2001) and virtual private network (VPN) (Gleeson et al., 2000), which are offered as standard services by mainstream cloud vendors for creating virtual networks in the cloud. A number of prior works adopt virtual networks in the geo-distributed environment to connect applications running in geo-distributed, isolated network domains and improve the network performance among them. Baldine et al. (2012) build the ExoGENI on top of a federation of geo-distributed cloud sites and network circuit providers, allowing users to create dedicated virtual networks for deeply networked, multi-domain, multi-site applications at scale. Cai et al. (2016) develop the cloud-routed overlay networks using VMs provisioned by cloud providers to re-route the Internet traffic and bypass the Internet core, avoiding the congestion at the Internet core and therefore improving the network performance over the Internet. Pfaff et al. (2009) and Pfaff et al. (2015) abstract the network switching and routing functionalities into a software component, further facilitating the creation and deployment of virtual networks on commodity servers and VMs.

Furthermore, the emergence of SDN advances network virtualization by enabling the programmability of networks and presenting a flexible network architecture on which complex algorithms and configurations are made possible. Casado et al. (2007) propose an early architectural design of the SDN of today. McKeown et al. (2008) extend it into the OpenFlow protocol, which evolves as a major SDN communication protocol of today. Gude et al. (2008) develop the first network OS implementing the OpenFlow protocol, which serves as the reference implementation for OpenFlow controller frameworks. Koponen et al. (2010) introduce a distributed network control platform for SDN networks.

Figure 2.3 (Braun and Menth, 2014) shows the SDN architecture. In traditional networks, the control layer is embedded in the physical switches together with the data layer; since the switches are autonomous and lack information about the network and applications, the control layer can only implement simple

decentralized algorithms to improve network performance in a best-effort manner. SDN abstracts the control layer into a centralized, stateful software component, referred to as an SDN controller, which can be fed with internal and external states of the network and uses them to drive complex routing algorithms. An SDN controller is equipped with a northbound and southbound APIs – the controller interacts with applications via the northbound API to acquire external states, and communicates with SDN-compatible switches via the southbound API to learn of the network states and install network flows. By centralizing the control layer, the SDN controller can gain a global view of the entire network and perform optimization for network routing. Additionally, with the northbound API, it is more knowledgeable about applications running on top than its counterpart in traditional networks.

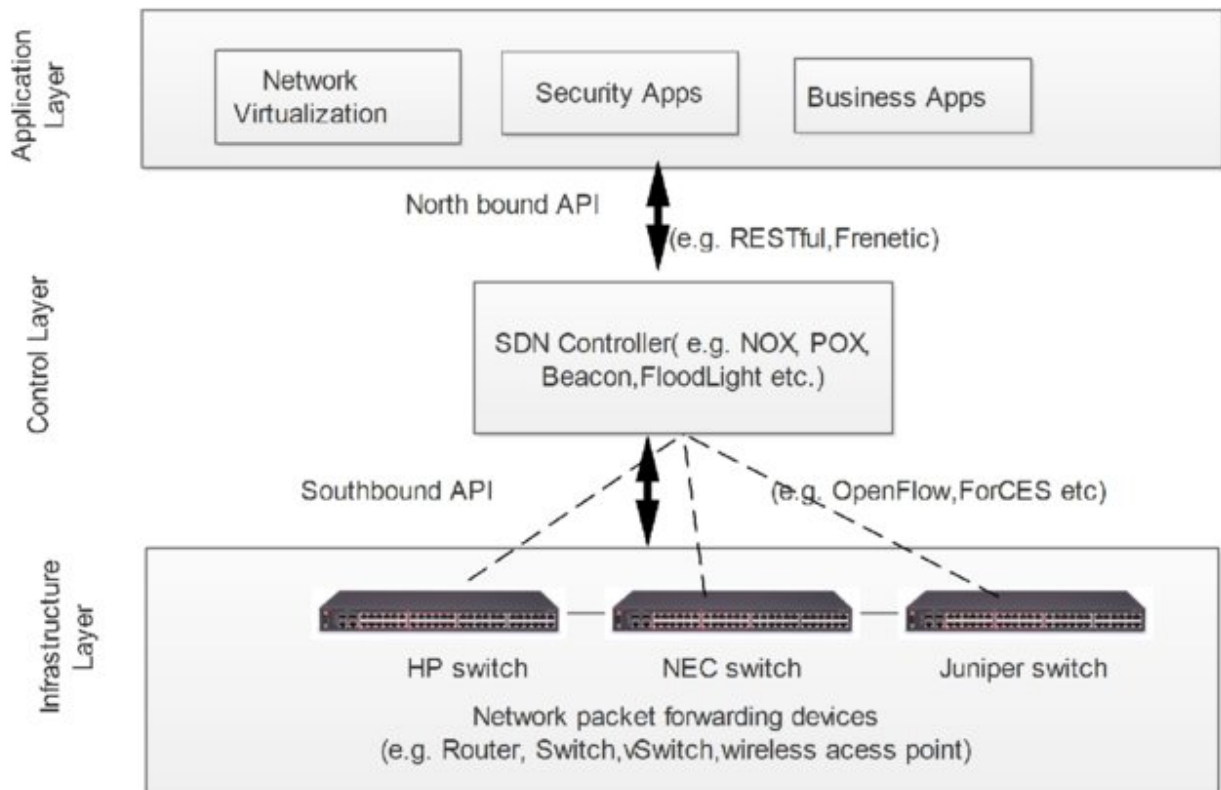


Figure 2.3: SDN Architecture (Braun and Menth, 2014)

The advance of SDN standards and technologies propels its adoption in academia and industry, such as GENI (Berman et al., 2014), ExoGENI (Baldine et al., 2012), B4 (Jain et al., 2013), SWAN (Hong et al., 2013), Jupiter (Singh et al., 2015), among others. It also stimulates the innovation of SDN-based network scheduling algorithms, which are impossible in the traditional network architecture without the network abstraction. For instance, Hong et al. (2013) model the bandwidth allocation among geo-distributed data

centers as a multi-commodity (MCF) problem and develop a global optimization approach to maximize the bandwidth utilization in WANs. In Chapter 3, we introduce our approach based on this work to address priority-based bandwidth allocation in WANs.

2.2 Data Locality in Distributed Computing

Data locality refers to the property of keeping computation in proximity to the data it needs. Preserving data locality is crucial for distributed data-intensive applications, since it avoids the significant performance overhead for massive data movement and therefore boosts application execution. Particularly, we treat the absence of data immediately needed by a computational task as the loss of data locality, since it costs extra time to retrieve the data by performing disk or network I/O or generate the data by executing associated computation. More importantly, preserving data locality for applications may also have financial implications, especially in the cloud, as cloud providers monetize the resources needed for storing and transferring data in the geo-distributed environment, and preserving data locality may avoid certain unnecessary cloud expenses.

Traditional HPC resource managers (Zhou, 1992; Staples, 2006) neglect the importance of data locality, since HPC clusters are equipped with high-speed LANs and storage area networks, and performance overhead caused by local data transmissions is trivial due to small data volume; meanwhile, these clusters rarely scale and thus incur predictable overhead for data movement. However, as the size of clusters scales and data volume grows, the overhead due to loss of data locality is magnified and causes significant performance degradation for data-intensive applications. Herein, there is a great amount of efforts committed to developing resource management mechanisms for preserving data locality for distributed applications. Raman et al. (1999) invent a formal language that describes custom node matching policies in Condor (Litzkow et al., 1987) clusters, which can be used for specifying explicit data locality constraints. However, the policies described in this language are relatively rigid and have limitations on expressing data locality requirements inclusively. Hindman et al. (2011) develop a two-level scheduling model in Mesos, which abstracts resources in a cluster in the form of *resource offers* at the lower level and delegates the application-specific task scheduling to the schedulers of application framework ported at the higher level. The advantage of this model is that it eliminates the coupling between application frameworks and underlying resources, allowing tasks to be scheduled freely in proximity to the needed data and therefore achieving data locality.

2.2.1 Computation Caching

A plethora of studies (Agrawal et al., 2008; Olston et al., 2008; Logothetis et al., 2010; Gunda et al., 2010) reveal that redundant computations are commonly performed within data centers, especially by data-parallel workloads, causing severe waste of computing resources and affecting data locality as local computational results cannot be reused but are unnecessarily re-computed. To tackle this problem, the concept of computation caching is proposed (Popa et al., 2009; Zaharia et al., 2010b) to reuse working sets of applications and avoid duplicate computation execution for improved data locality. Chambers et al. (2010) defer evaluation of data-parallel pipelines to seek for opportunities for reusing existing computational results. Zaharia et al. (2012) identify that computation caching serves well for iterative (e.g., machine learning algorithms) and interactive (e.g., data analyses) workloads in data-intensive applications and develop a fault-tolerant, in-memory data structure that retains working sets for reuse within applications. Gunda et al. (2010) and Li et al. (2014) break the boundary between applications and allow the in-memory data sets to be shared across applications. Gunda et al. (2010) implement computation caching by caching intermediate data of application jobs in a central server and employing a utility function for cache replacement. Rabkin et al. (2014) first extend this concept into the geo-distributed environment to cache intermediate results of data analytics queries and accelerate repeated queries initiated by geo-distributed clients.

However, existing computation caching approaches only adopt simplistic caching strategies, e.g., least-recently-used (LRU) (Zaharia et al., 2012), which are insufficient for environments with a high level of heterogeneity, e.g., the geo-distributed environment (Pu et al., 2015). On the other hand, there are plenty of complex data caching and replication strategies invented for geo-distributed environments. Herein, exploring existing caching and replication strategies may be conducive to improving computation caching in geo-distributed environments.

2.2.2 Distributed Data Caching and Replication

The ultimate goal of caching is to capture the most reusable data with limited cache space and speed up applications by reusing the cached data. It is particularly effective for workloads with a long-tail distribution, which are prevalent in geo-distributed environments. Glassman (1994) first recognizes that web requests follow the Zipf's distribution (Zipf, 1929) and concludes that caching frequently used web pages reduces user-perceived latency. Later studies (Breslau et al., 1999; Nygren et al., 2010; Fricker et al., 2012; Shafiq

et al., 2014) confirm that the Zipf's distribution is extensively applicable to modeling data distribution and access pattern in the geo-distributed environment. This insight provides important guidance for designing effective and generalizable distributed caching strategies and algorithms.

As suggested by Fricker et al. (2012) and Shafiq et al. (2014), LRU, least-frequently-used (LFU), and their minor variants are commonly used in geo-distributed environments for their simplicity and stability. Cao and Irani (1997) extend the LFU algorithm by factoring data size into cache replacement decisions and favoring the retention of small-sized data objects in cache. Rizzo and Vicisano (2000) propose the least-relative-value (LRV) algorithm in which a utility function is used to evaluate the relative values of data objects in cache and ones with the least relative value will be evicted from a full cache. Lee et al. (2001) combine the LRU and LFU algorithms to handle both recently and frequently used data objects in a weighted manner. Gunda et al. (2010) devise a novel utility function that factors in the cost for generating data objects by executing associating computational jobs, which is unique in the scenario of computation caching. However, Shafiq et al. (2014) argue that using these simplistic algorithms is insufficient for effective data caching in the geo-distributed environment considering the unique workload characteristics and geo-distributed system settings. This argument is supported by our experimental results shown in Chapter 4. Hence, it is compelling to investigate more delicate caching strategies and algorithms suitable for geo-distributed data caching.

Cooperative caching is considered as an efficacious caching strategy capable of fully utilizing geo-distributed cache storage and co-locating data in proximity to their consumers. In general, cooperative caching caches popular data sets in the cache space contributed by distributed nodes in a system and coordinates the distributed cache to make them function as a unified, gigantic global cache. This strategy is widely adopted in distributed systems at varying scales (Chand et al., 2007; Cao et al., 2007; Cho et al., 2012a). For instance, as indicated by Nygren et al. (2010), Akamai adopts a form of cooperative caching to cache popular content at the geo-distributed edges for rapid content rendering. Dahlin et al. (1994) first introduce this strategy to distributed file systems, caching popular files in the memory of distributed clients to improve the system responsiveness. However, they omit the discussion on data replica management, which is crucial in cooperative caching since data duplication is inevitable, and unregulated data replication may cause waste of valuable cache space. Surlas et al. (2013) focus on managed data replication in cooperative caching, proposing online, autonomic algorithms that makes intelligent decisions on data replication and replica placement. Ming et al. (2014) develop a lightweight, age-based collaboration mechanism for cooperative caching, reducing the overhead for coordination of distributed cache.

As a complementary measure to data caching, data replication creates multiple replicas of popular data sets across geo-distributed data stores, trading storage space for avoidance of expensive remote data transfers over WANs. Although few works integrate geo-distributed data caching and replication, there is a bulk of literature on geo-distributed data replication. Kangasharju et al. (2002) formulate data object replication in content distribution networks as a combinatorial optimization problem and propose heuristics for approximating the optimal replication plans. Chervenak et al. (2002) build a scalable metadata service that keeps track of locations of geo-distributed replicas. This work inspires our development of the logically centralized metadata service for CACHALOT as introduced in Chapter 4. Corbett et al. (2013) introduce Google’s globally distributed database named Spanner, which implements synchronous data replication at the global scale. Gupta et al. (2014) present Google’s geo-replicated data warehousing system named Mesa, which serves billions of data analytics queries in a near real-time manner. Muralidhar et al. (2014) develop a large-scale, geo-replicated storage system for storing binary large objects (BLOBs) in Facebook, and focus on the mechanism for dynamically tuning the replication factor. Wu et al. (2013) construct a cost-effective, geo-replicated key-value store spanning across multiple cloud platforms.

Last but not least, network factors have a non-trivial impact on the performance of geo-distributed caching due to frequent data transmissions over WANs among the distributed cache. Rizzo and Vicisano (2000) develop a network-aware cache algorithm based on the assumption that the data transfer rate is uniform throughout the network, which is unrealistic in WANs. Kalnis et al. (2002) build a peer-to-peer cache network that relies on static, coarse-grained network metrics to capture network factors at low accuracy. Borst et al. (2010) and Sourlas et al. (2013) take optimization approaches to incorporate network factors for optimal cache management decisions. However, it is debatable whether these heavyweight approaches are scalable.

2.2.3 Locality-Aware Scheduling

Opposite to data caching and replication, locality-aware scheduling gains data locality by placing computation in proximity to data it needs. This approach has been thoroughly studied in data-parallel computing and event processing systems.

Isard et al. (2009) introduce a locality-aware scheduling algorithm for Dryad (Isard et al., 2007) that optimizes for data locality and fairness requirements imposed on applications using the min-cost flow algorithm. Based on the optimal algorithmic result, the algorithm preempts and relocates tasks in progress, wasting the work done by the preempted tasks and potentially delaying their completion. Despite the optimal

task placement for data locality, this intrusive approach can be detrimental to applications since the speedup gained from improved data locality may be overrun by the delay caused by task preemption. Instead, Zaharia et al. (2010a) propose to intentionally delay the scheduling of tasks that cannot launch immediately on nodes with needed data for a predetermined amount of time, expecting the desired nodes to be available shortly during the waiting and therefore preserving data locality. This approach passively co-locates tasks with data and retains data locality. However, it is only feasible for workloads with mostly short tasks, in which case the delay scheduling will be rewarded by improved data locality but defers task execution otherwise. Instead of the static data locality constraint, Boutin et al. (2014) develop a function that gradually relaxes the data locality constraint as the delay of task scheduling increases to accommodate data locality and scheduling timeliness dynamically. Similarly, Jonathan et al. (2016) introduce a minimum locality level for every task that allows a task to be scheduled on time with suboptimal but acceptable data locality, trading the optimality of data locality for timely task scheduling.

Locality-aware scheduling has also been adopted in geo-distributed environments to reduce the costs for bandwidth usage while improving performance by limiting inefficient data transfers over WANs. Hung et al. (2015) concentrate on the coordination of tasks among geo-distributed data centers to gain data locality. They realize that skewed distribution of task completion time among data centers tends to prolong job execution, since job completion time is determined by the last finished task. To address the problem, the authors propose to reorder tasks among data centers with respect to their completion time in order to eliminate outliers and balance task completion time across data centers. They also propose a greedy scheduling heuristics that prioritize the scheduling of the shortest tasks, which significantly improves the average job completion time. Pu et al. (2015) and Viswanathan et al. (2016) focus on network factors in geo-distributed environments, which greatly impact application performance. Pu et al. (2015) propose to jointly schedule tasks and data to achieve data locality for data analytics queries in geo-distributed Spark deployment. The authors develop an online heuristic that redistributes data sets and co-locates tasks with the data to circumvent network bottlenecks during query execution. Viswanathan et al. (2016) build WAN awareness into query optimizers for data analytics, which make joint decisions on the optimal query execution plan that places tasks in proximity to data in consideration of WAN factors.

The aforementioned approaches are based on the common assumption that the locality of data is clearly known to applications. This assumption rarely holds true for applications that process dynamic data, e.g., data streaming applications, in which data processing components are deployed before ingesting data.

Therefore, these applications must infer data locality in order to optimize task scheduling. For instance, Peng et al. (2015) adopt a bin-packing approach that greedily consolidates processing operators of every data streaming application into a few machines in a cluster to increase the chance to co-locate interacting operators. However, this approach packs operators based on resource demand and availability with little insight into communications between operators. In contrast, Xu et al. (2014) monitor network traffic between operators and dynamically co-locate those with heavy traffic exchanges. Caneill et al. (2016) take a data-oriented approach, uncovering correlations between keys processed by successive operators and using the correlations as evidence to guide the co-location of operators.

2.2.4 Data Locality and Cost Awareness in the Cloud

Only preserving data locality is insufficient for data-intensive applications running in geo-distributed environments, especially in the cloud, since cloud vendors impose charges on resources for running data-intensive applications, and there are price discrepancies among different types of resources and providers. Without the awareness of costs associated with resource subscription, running data-intensive applications in the cloud can incur substantial cloud expenses unnecessarily and increase the financial burden. Further, retaining data locality is mostly aligned with saving cost in the cloud, since it deters unnecessary VM subscription and remote data transfers that are heavily charged by cloud vendors. In the meantime, it entails certain costs necessary for co-locating computation and data. Hence, it is crucial to preserve data locality while gaining cost awareness for data-intensive applications in the cloud.

Remote data transfers contribute a major portion of cloud expenses since they tend to incur egress network traffic, which is the traffic propagated across different geo-locations or over the Internet and heavily charged by mainstream cloud providers. Hence, avoiding unnecessary remote data transfers is conducive to reducing cloud expenses as well as maintaining data locality. To save the cost for egress network traffic, Wu et al. (2013) gain insight into disparate charge rates of egress network traffic among cloud regions and providers and exploit the discrepancies to select most cost-effective geo-locations and cloud platforms for data replication. Pu et al. (2015) enable a mechanism for budgeting WAN usage, trading off data locality and WAN usage expenses by limiting the amount of WAN bandwidth used for data redistribution and task execution to a predetermined budget. Kloudas et al. (2015) introduce an auxiliary topology abstraction for data-parallel workflows that groups nodes in a cluster by their geo-locations. The abstraction facilitates the task scheduler to assign closely communicating tasks of a workflow to nodes at the same geo-locations in

order to minimize data transfers over WANs and restrict costs for WAN usage. Hsieh et al. (2017) propose a cost-effective, geo-distributed machine learning model that can converge with approximately synchronous parameters and eliminates unnecessary cross-region communications, significantly saving the monetary cost associated with parameter synchronization across geo-distributed data centers.

In addition, VM subscription costs are also a major source of cloud expenses. Traditional elastic computing solutions adopt the one-task-per-instance model (Mao et al., 2010; Van den Bossche et al., 2010; Bittencourt and Madeira, 2011; Mao and Humphrey, 2011) to attain dynamic resource provisioning in the cloud, which harms data locality since inter-task communication is limited to inter-node data transmission and therefore slows down application execution. Further, the coarse-grained task scheduling model tends to result in low utilization of subscribed VM instances, especially for data-intensive workloads, as the predominant IO-bound tasks can hardly saturate the capacity of every single VM but wastes computing resources allocated with each VM instance. Chung et al. (2018) and Gunasekaran et al. (2019) propose to adopt lightweight task encapsulation with containers and serverless functions, respectively, opening the door to packing multiple tasks into a single VM and saving VM subscription cost. Specifically, Chung et al. (2018) develop a cost-aware task scheduler that dynamically scales the VM cluster at a fine granularity of VM size and consolidates tasks in the form of containers into VM instances in the cluster based on the estimation of task runtime. The task consolidation technique used by Chung et al. (2018) stems from the general multi-dimensional vector bin packing (MDVBP) problem, which has been thoroughly studied in energy-conserving task scheduling (Li et al., 2009; Beloglazov and Buyya, 2010, 2012; Knauth and Fetzer, 2012) and VM packing (Sindelar et al., 2011; Corradi et al., 2014; Ahmad et al., 2015). The optimization for this problem is proven NP-hard (Frenk et al., 1990). Panigrahy et al. (2011) conduct an inclusive survey on a variety of approximate heuristics for the MDVBP problem.

2.3 TCP Optimization for Remote Data Transfer

Despite preservation of data locality, the efficiency of remote data transfers in WANs is vital to the performance of data-intensive applications, since they are inevitable in geo-distributed environments. Importantly, TCP/IP is still the dominant Internet protocol suite enabling remote data transfers. However, since initially designed for LANs, TCP exhibits its shortcomings in WANs which cause serious performance deterioration. Lakshman and Madhow (1997) recognize that the classic TCP design (Jacobson, 1988) discriminates against

connections with high latency and overreacts to random packet loss. This behavior is unfriendly to remote data transfers as they typically show high latency and the tendency of packet loss. Further, the authors reveal that large product of bandwidth and network delay on a TCP connection, referred to as bandwidth-delay product (BDP), may overflow buffers of network devices along the network path and further deter TCP performance with increasing packet loss. Mathis et al. (1997) manifest the problem by formulating a mathematical model to describe the TCP fast retransmit mechanism as follows:

$$BW = \frac{MSS \cdot C}{RTT \cdot \sqrt{p}} \quad (2.1)$$

In this equation, BW , RTT , and p denote bandwidth, round trip time (RTT), and the probability of packet loss, respectively. MSS and C are constant values, representing maximum segment size (MSS) and constant of proportionality. As reflected in the equation, the probability of packet loss is inversely correlated with bandwidth and RTT. In other words, high BDP will amplify packet loss, which in turn hurts throughput and degrades TCP performance. Padhye et al. (1998) refine the model of Mathis et al. (1997) by incorporating the TCP timeout mechanism. Floyd et al. (2000) propose the equation-based congestion control as a replacement for the best-effort, additive-increase/multiplicative-decrease (AIMD) mechanism using such mathematical models, which adjusts the sending rate as a function of measured rate of packet loss.

Centering on the macroscopic behavior of TCP congestion control, an array of TCP variants are developed to cope with performance degradation observed in long-distance data transfers in WANs. The TCP Westwood (Mascolo et al., 2001) focuses on TCP congestion control for wireless networks with lossy links. It introduces a fast recovery mechanism that readjusts the congestion window consistently with the sender-side bandwidth measurement, in contrast with blindly halving the congestion window as performed by the TCP Reno, upon packet loss. In effect, it reduces the false positives of congestive packet loss and therefore alleviates the disturbance of packet loss to TCP throughput. The TCP Veno (Fu and Liew, 2003) adopts a similar approach to distinguish between congestive and random packet loss in wireless networks. Leith and Shorten (2004) devise the H-TCP for high-speed, homogeneous networks – they observe that the additive increasing factor in a high-speed, homogeneous network diverges from that in a conventional network, and they dynamically adjust the factor to accommodate different network settings and fully utilize available bandwidth. Xu et al. (2004) and Ha et al. (2008) propose to accelerate the additive increasing phase to overcome the lag in congestion window recovery caused by high network delay.

The prevailing approaches share a common assumption that packet loss is an indication of network congestion. The rationale behind this is that network congestion causes bandwidth oversubscription, and certain packets are forcibly dropped due to buffer overflow on intermediate network devices. However, as network capacity exponentially grows, the relationship between packet loss and network congestion becomes tenuous as identified by Cardwell et al. (2016). Hence, Cardwell et al. (2016) build a non-loss-based congestion avoidance model based on bandwidth and RTT and devise a new distributed congestion control mechanism named BBR which achieves optimal bandwidth utilization as a result.

An alternative to tackling TCP performance degradation caused by large BDP is to use multi-path TCP (MPTCP) initially proposed by Ford et al. (2013). The core idea is to perform a data transfer using parallel TCP sub-flows simultaneously to alleviate the negative impact of high latency and packet loss – congestion window growth is multiplied by parallel TCP sub-flows, while slowdown caused by packet loss is diluted among the multiple TCP sub-flows. This approach is commonly adopted by data transfer tools for long-distance data transfers. Raiciu et al. (2012) provide an in-depth explanation of the MPTCP implementation.

CHAPTER 3: Advancing Data-Driven Scientific Collaboration in Geo-Distributed Environment

In this chapter¹, we present our work in the RADII project that facilitates data-driven scientific collaborations among geo-distributed research institutions. These scientific collaborations are common, especially in multidisciplinary projects, in which domain scientists are committed to specific areas but put forward the project collaboratively. Typically, the collaborations are realized in the form of data-intensive applications such as data processing workflows and pipelines consisting of computational jobs running in a geo-distributed fashion among the participating institutions, which desperately call for streamlined infrastructure support in the geo-distributed environment.

The advance in national CI and cloud computing has dramatically improved the accessibility of physical infrastructure in the geo-distributed environment. However, domain scientists still face the prevalent challenges in geo-distributed systems when deploying and executing data-intensive applications for scientific collaborations. Mainly, they lack the high-level abstraction and mechanism for describing, constructing, and managing such collaborations with simplicity and accuracy. Furthermore, data sharing among geo-distributed locations tends to create the significant performance bottleneck due to the absence of network abstraction and optimization.

Hence, we have developed RADII as an integrated solution to address these challenges. We have introduced a simple yet powerful data model for describing scientific collaborations. We have also built the software infrastructure that maps the high-level collaboration description to low-level, virtualized infrastructure primitives for deploying and orchestrating the collaboration on the physical infrastructure at a fine granularity. In particular, we use software-defined networking (SDN) to abstract the network. On top of that, we have devised an MCF-based optimization approach and developed a TCP enhancement to improve the bandwidth utilization and accelerate remote data transfers in the collaboration.

¹Content of this chapter previous appeared in preliminary form in the following paper:
Jiang, F., Castillo, C., and Schmitt, C. (2016). RADII: Bridging the Divide Between Data and Infrastructure Management to Support Data-Driven Collaborations. In *Big Data (Big Data)*, 2016 IEEE International Conference on, pages 370–377. IEEE.

For the rest of this chapter, we first introduce the high-level data model (Section 3.1) and the software infrastructure (Section 3.2) that support the scientific collaborations. Then, we dive into the SDN-based network abstraction (Section 3.3) and explain the MCF-based network optimization (Section 3.4) and TCP enhancement (Section 3.5) in detail. Lastly, we present the experimental evaluation (Section 3.6) and summarize this chapter (Section 3.7).

3.1 Collaboration Representation

3.1.1 Scope

First, we define the scope of the *collaboration* discussed in this chapter. At a high level, a *collaboration* is a type of scientific project in which researchers from different institutions, referred to as *collaborators*, work together by sharing the computational workloads and related data sets to achieve the common goal of the project. In detail, it embodies a large-scale data processing application developed and executed collaboratively by the *collaborators*, which consists of data processing and analytical jobs running upon raw data hosted at the geo-distributed institutions. A *collaboration* also encapsulates a full-fledged, tailored infrastructure on which the *collaborators* can develop, deploy, and execute the data processing jobs, and share data among them. Furthermore, a *collaboration* enforces data policies to protect data from unauthorized personnel. In a nutshell, a *collaboration* is a comprehensive, end-to-end encapsulation of a distributed system for collaborative works among multiple institutions. Consequently, it is a non-trivial task to describe a *collaboration* with clarity and accuracy.

3.1.2 Collaboration Model

To assist with the description of *collaborations*, we have developed a data model based on the DFD, referred to as the *Collaborative Language User interface (CLUE) object*, which encodes the core artifacts in a *collaboration* including the *collaborators*, *data policies*, and the *data flow* (left box in Figure 3.1).

3.1.2.1 Collaborator

A *collaborator* represents a uniquely identified user who participates the *collaboration*. Each *collaborator* owns assets related to authentication and authorization, among which the most important ones are the public

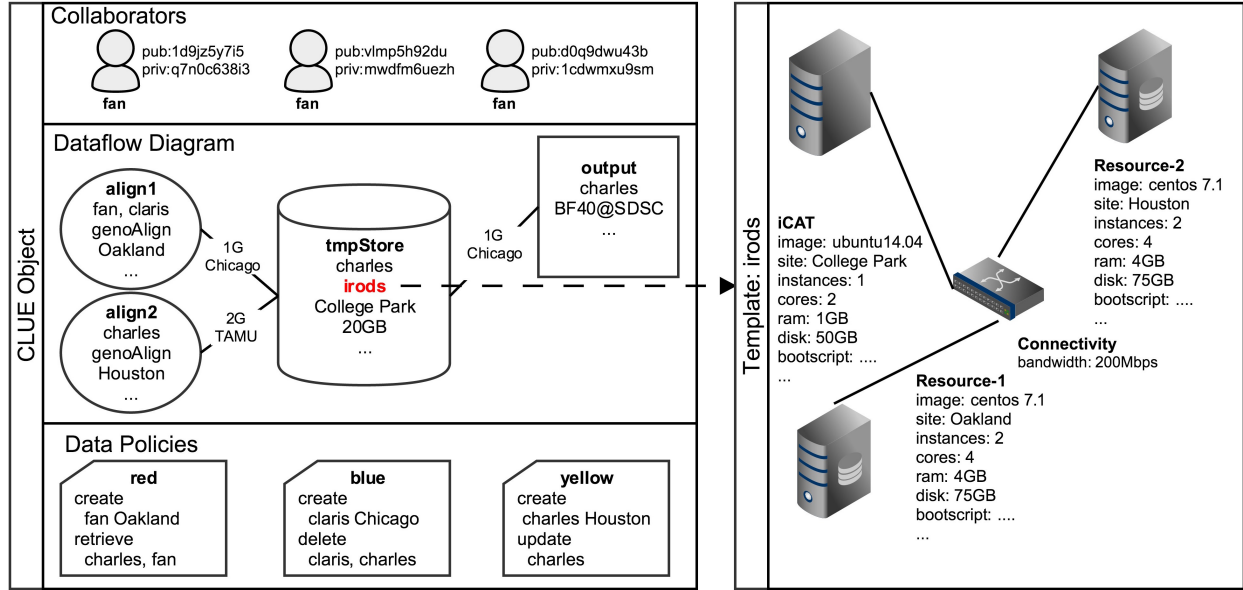


Figure 3.1: CLUE Object

and private keys. The key pair is used for a wide range of activities throughout the *collaboration*, e.g., data encryption and decryption, access authentication, among others.

3.1.2.2 Data Policy

The *data policies* authorize the *collaborators* for their operations upon data sets in the *collaboration*. Each *data policy* has a unique identifier called *tag*. To protect a data set with a data policy, the authorized *collaborator* associates its *tag* with the data set, so that the rules specified in the policy can be enforced. Each *data policy* functions as the POSIX access control list (ACL), comprising a list of rule entries in the form of $\langle \text{operation}, \text{collaborator}, \text{location} \rangle$, which is interpreted as the operation is permitted to perform by the *collaborator* upon the protected data set if it is stored at the *location*. For the preliminary implementation of *data policies*, we only consider the basic CRUD (Create, Retrieve, Update, and Delete) operations to realize the rudimentary access control. Enhancing the *data policies* with a more sophisticated scheme, although highly feasible, is out of the scope of this dissertation.

3.1.2.3 Data Flow

The *data flow* is captured in a DFD, which consists of four types of entities as illustrated in Table 3.1. Each entity encompasses the software and hardware resources related to it, which are embedded as its attributes in the entity.



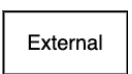

Legend	Name	Description	Key Attributes
	process	One or a group of data processing jobs	cpus, ram, location, anti-locations
	datastore	A data storage protected by <i>data policies</i>	capacity, location
	external	An external data source or repository	url, credentials
	dataflow	A network path connecting a pair of entities for data transfers	bandwidth, anti-providers

Table 3.1: DFD entities

A *process* represents one or a group of homogeneous data processing jobs in a *collaboration*, which mainly specifies the software stack (e.g., program, runtime, OS) and computing capacity (e.g., number of CPUs and GPUs, RAM size) for executing the jobs. It usually serves an intermediate entity in the *data flow*, consuming input data from the upstream entities and producing output data for the downstream ones. Further, it has a `location` attribute, which indicates the geo-location where the *process* will be performed. Under the hood, this attribute determines the physical infrastructure at a specific geo-location where the data processing jobs will be deployed and executed. With this attribute, a *collaborator* can place a *process* in a preferred physical location, which is close to the input data or in an authorized domain. Beyond that, a *process* also has an `anti-locations` attribute that indicates the geo-locations to avoid, which can signal the underlying scheduling mechanism to skip resources at the specified locations.

A *datastore* specifies the data storage where the *data policies* are enforced. Each *datastore* is associated with one or several data policies defined for the *collaboration*, which protect the data stored inside. A *datastore* typically stores the input and intermediate data shared among the *processes*. Like the

process, it also has an explicit `location` attribute that determines the geographical placement of the underlying storage unit, which can be used by the *collaborators* to control the physical data placement.

An *external* is a data source or repository external to the *collaboration*, which is accessible but not controlled by any *collaborator*, e.g., Google Drive, Dropbox. It works as a black box for other entities to read and write data, typically serving input data and archiving the final output data for a *collaboration*. An *external* is identified by the URL, which is invoked by other entities in the *collaboration* to read or write data.

A *dataflow* defines a bidirectional network path connecting a pair of entities for data transfers. It has an attribute `bandwidth` for setting the bandwidth requirement on the path, which can be used to throttle network traffic between entities. Optionally, *collaborators* can also specify the `anti-providers` on a *dataflow*, which are the network providers the network path will bypass when being deployed.

3.1.2.4 Template

With the high-level artifacts, researchers can easily initiate a *collaboration* without diving into the technical details. However, they are barely omitted but embedded as the attributes of the DFD entities instead. In particular, describing a complex *collaboration* is still laborious to properly address the numerous attributes. Hence, we introduce *templates* (Figure 3.1) for the DFD entities to further simplify the description of *collaborations*.

A *template* is a DFD entity of which the attributes are pre-populated with sane default values. It usually maps to a commonly used `process` or `datastore` and can be directly instantiated without any modifications. With *templates*, a researcher can create a DFD entity by selecting a needed *template* from a list of pre-built ones and only modifying the values of attributes of interest, rather than building it from scratch. Furthermore, the pre-populated attributes in *templates* provide extra guidance to construct the entity properly.

3.1.3 Collaborative Infrastructure

A *CLUE object* comprises the software and hardware specifications for a *collaboration* and will eventually be instantiated on the underlying infrastructure. This physical instance forms a virtual infrastructure dedicated to the *collaboration*, in which the software and hardware resources are deployed and configured according to the *CLUE object* to serve the *collaboration*. Moreover, the virtual infrastructures of different *collaborations*

are mutually isolated to prevent potential security loopholes and performance interference. We refer to such infrastructure as *collaborative infrastructure* (Figure 3.2).

The implementation of a *collaborative infrastructure* is loosely coupled with the underlying computing platform - despite the huge IT investment and performance overhead, traditional on-premise campus clusters can serve as a *collaborative infrastructure* if they can implement the functionalities specified in the *CLUE object*. However, IaaS offered by national CIs and cloud computing platforms, as introduced in Section 2, is preferable because of its elasticity, high availability, and programmability – it provisions resources elastically based on the varying demand in a *collaboration*; resources are rarely exhausted and can be instantly replaced upon failures; and most mainstream IaaS platforms provide API for automatic resource subscription and management. These merits of IaaS greatly smooth the construction and maintenance of a *collaborative infrastructure*. In RADII, we have built our system on top of the networked IaaS platform named ExoGENI² (Baldine et al., 2012) for its worldwide accessibility and versatility in resource provisioning.

In this section, we elaborate on the data model for the formal representation of *collaboration* and its embodiment at the infrastructure level. In the next section, we introduce the system design of RADII, which orchestrates the translation, deployment, and management for *collaborations*.

3.2 System Design

RADII serves as the middleware that translates a high-level *collaboration* representation into a low-level *collaborative infrastructure* and orchestrates its deployment and management on the underlying computing platform. It also deploys an SDN network for each *collaborative infrastructure*, lending the opportunity for custom network optimization.

The system design of RADII follows the service-oriented architecture (SOA), composed of software components in the form of loosely coupled services that interact with each other through their defined REST APIs. As illustrated in Figure 3.2, RADII consists of four major services: *CLUE*, *Orchestrator*, *Compute To Infrastructure (C2I)*, and *Data To Infrastructure (D2I)*.

²ExoGENI testbed: <http://www.exogeni.net>

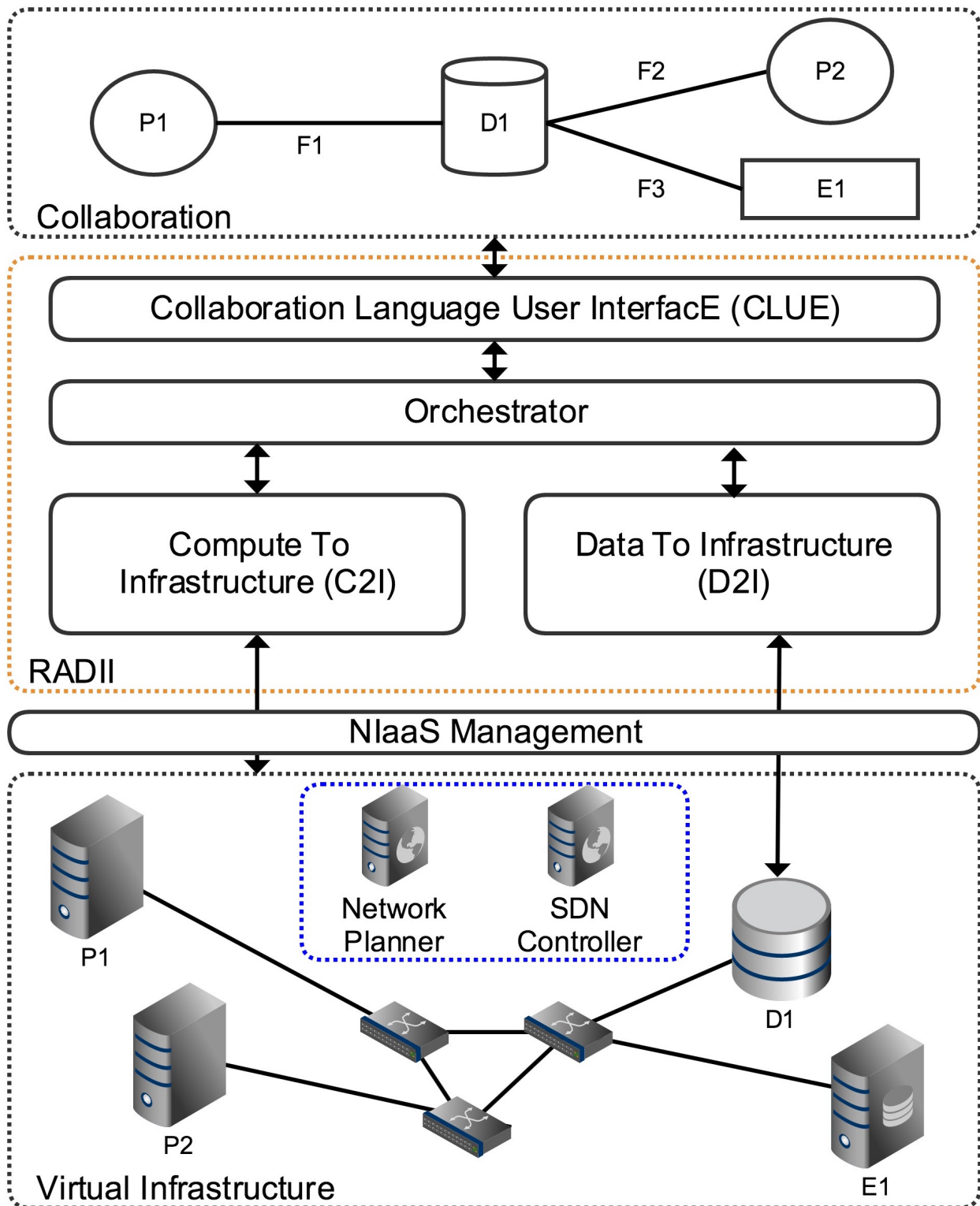


Figure 3.2: RADII architecture

3.2.1 CLUE

The *CLUE* is the user interface for creating and interacting with a *collaboration*. It serves as the external API of RADII, exposing a rich set of endpoints for composing and manipulating a *collaboration* programmatically. The *CLUE* uses the *CLUE object* introduced in Section 3.1 to represent a *collaboration*, which reflects the current state of the *collaboration*. The API streamlines the integration of RADII with other systems; however, it is less user-friendly since interacting with a *collaboration* directly through the API can be obscure and error-prone for end users. Hence, we have developed a web-based graphical user interface (GUI) as an auxiliary part, as shown in Figure 3.3, to provide an animated, drag-and-draw dashboard for users to interact with a *collaboration* with clarity.

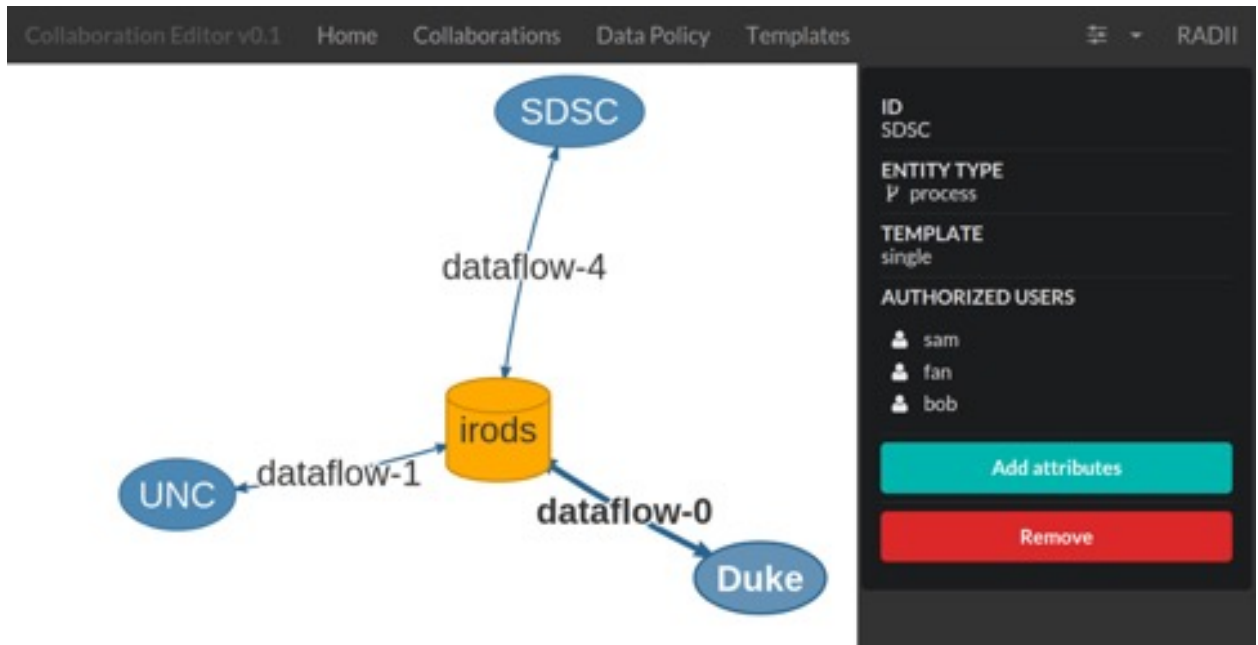


Figure 3.3: Web GUI of CLUE

3.2.2 Orchestrator

The *Orchestrator* orchestrates every activity and event related to a *collaboration* throughout its lifetime from creation to termination. Its chief responsibility is to coordinate between the computing- and data-related activities and events in a *collaboration*. Upon receipt of a *CLUE object*, the *Orchestrator* recognizes the computing- and data-related activities and convert them into requests consumable by the *C2I* and *D2I* downstream; when notified of any event by *C2I* or *D2I*, it updates the corresponding *CLUE object* of the

collaboration and sends a notification to the end user through *CLUE*. The *Orchestrator* is a stateful service that keeps track of the latest *CLUE objects* of *collaborations*. Since it is a key service connecting other components in the system, it is crucial to guarantee its high availability and reliability. Hence, by design, we decouple the database for storing *CLUE objects* from the *Orchestrator* to ensure its scalability and avoid the single point of failure. Thus, multiple instances of the *Orchestrator* can be launched in parallel on top of the same database, and failure of any instance, if not all, will not lead to a breakdown of the entire system. For the reliability and availability of the database, we resort to the known fault-tolerance solutions, such as Paxos (Lamport, 1998) and Raft (Ongaro and Ousterhout, 2014), and their implementations in data storage, such as Apache ZooKeeper³ and etcd⁴.

3.2.3 C2I

The *C2I* sets up the *collaborative infrastructure* for a *collaboration* as specified in the *CLUE object*. It has the logic built in for mapping the high-level artifacts in the *CLUE object* to specific resources provisioned by the underlying computing platform. Typically, with an IaaS platform underneath, *C2I* creates an array of VMs for a *process*, which are loaded with a self-contained VM image pre-configured with the software stack for running the specified data processing job; for a *datastore*, it deploys a data management system for storing and sharing data sets used in the *collaboration*; for an *external*, it spins up a lightweight API in front of the corresponding external resource to ensure other resources in the infrastructure being able to communicate with it; for a *dataflow*, it sets up a network path between the specified ends, on which the bandwidth can be reserved as specified.

The *C2I* communicates with the underlying platform through its standard API to subscribe, query, and control the resources. Particularly, for IaaS platforms, instruments such as the Network Description Language (NDL) (Van der Ham et al., 2006), Apache Libcloud⁵, and Terraform⁶ are developed to streamline the use of the API and facilitate the integration between *C2I* and the platforms. Besides, the *C2I* also interacts with certain resources in the infrastructure to bootstrap and maintain subsystems, e.g., data management

³Apache ZooKeeper: <https://zookeeper.apache.org>

⁴etcd: <https://github.com/etcd-io/etcd>

⁵Apache Libcloud: <https://libcloud.apache.org>

⁶Terraform by HashiCorp: <https://www.terraform.io>

subsystem, using DevOps tool stack such as Ansible⁷ and SaltStack⁸. It is worth noting that the *C2I* is unnecessarily bound to a single computing platform but able to span across multiple platforms to acquire more diversified and geo-distributed resources. We will expand on this potential in Chapter 5.

Notably, the *C2I* also creates an SDN virtual network for each *collaborative infrastructure* to abstract the network layer, which provides extra control of the network and thus opens the door to potential network optimization for data sharing within the *collaboration*. We will elaborate on the design and mechanism of the virtual network in Section 3.3.

3.2.4 D2I

The *D2I* is responsible for translating and enforcing the *data policies* specified for a *collaboration*. It translates the *data policies* into data security rules compatible with the data management system underpinning the *datastore*, and installs the rules onto the data management system by invoking its API to secure the data sets stored inside.

As with the *C2I*, *D2I* is barely bound to any data management system underneath. However, as data is produced and consumed by geo-distributed jobs in most *collaborations*, the underlying data management system must be a distributed storage system, which can store and manage data in proximity to the jobs in a geo-distributed fashion. The mainstream distributed storage system, such as Network File System (NFS) (Sandberg et al., 1985), Ceph (Weil et al., 2006a), and the Hadoop Distributed File System (HDFS) (Shvachko et al., 2010), implement the POSIX ACL or alike, which suffices the basic access control with simplicity. For advanced access control, we can either introduce a shim layer implementing the sophisticated access control mechanism in front of the data management system, e.g., Lightweight Directory Access Protocol (LDAP) (Sermersheim, 2006), or opt for a distributed storage system with the advanced data security mechanism built in, e.g., iRODS (Rajasekar et al., 2010).

In our implementation, we select the iRODS as the data management system underpinning *collaborations* for its versatile rule engine, which provides a rule language resembling a full-featured programming language that allows users to define complex rule procedures to protect data at fine granularity. Specifically, in RADII, the *datastores* in a *collaboration* are deployed as an iRODS data grid consisting of a metadata server, which controls the access to any data stored in the grid, and a number of storage servers, which have the

⁷Ansible: <https://www.ansible.com>

⁸SaltStack: <https://www.saltstack.com>

actual data stored inside. The *D2I* interprets the *data policies* embedded in a *collaboration* to formulate the iRODS rule procedures accordingly, which are then installed on the metadata server to enforce the *data policies* and safeguard the data.

3.3 SDN-Based Network Abstraction

Massive data transfers among geo-distributed resources can be extremely slow, causing unpredictably long delay for data-intensive applications. The culprit of the slowness is mostly the inefficiency of the underlying WAN, which may have limited bandwidth, high latency, and severe network congestion. This problem also recurs in scientific collaboration, in which large data sets are commonly shared over the WANs, if not the Internet. However, the rigidity of the traditional network architecture, as explained in Section ??, restricts the access to and control of the network infrastructure, and thus makes it impossible for researchers to improve the WAN performance to accelerate data transfers in the collaboration. Hence, we introduce a network abstraction layer in the *collaborative infrastructure* by creating an SDN virtual network over the geo-distributed resources, to grant more control of the network to applications and make it possible to customize network optimization schemes for different *collaborations*.

Figure 3.4 depicts the network architecture in a *collaborative infrastructure*. At the bottom, the *physical network* lays the groundwork to provide the core connectivity among distributed resources. On top of it, there is a flat *virtual network* dedicated to the specific *collaboration*, routing network traffic among the resources subscribed in the *collaborative infrastructure*. Above that, the *network control plane*, which is programmed with the custom network routing algorithm, directs the network traffic in the *virtual network*.

3.3.1 Physical Network

The *physical network* is the foundation of the network system in a *collaborative infrastructure*. It can consist of a single network or a federation of multiple networks of the underlying computing platforms. Specifically, in the context of cloud computing, the *physical network* can be a federated network composed of geo-distributed virtual private clouds (VPCs) provisioned by multiple cloud vendors as we will introduced in Chapter 5. In general, the *physical network* is the network infrastructure that supports the fundamental networking capabilities in Layer 1 and 2, which underpin the functionalities enabled in the *virtual network* on top.

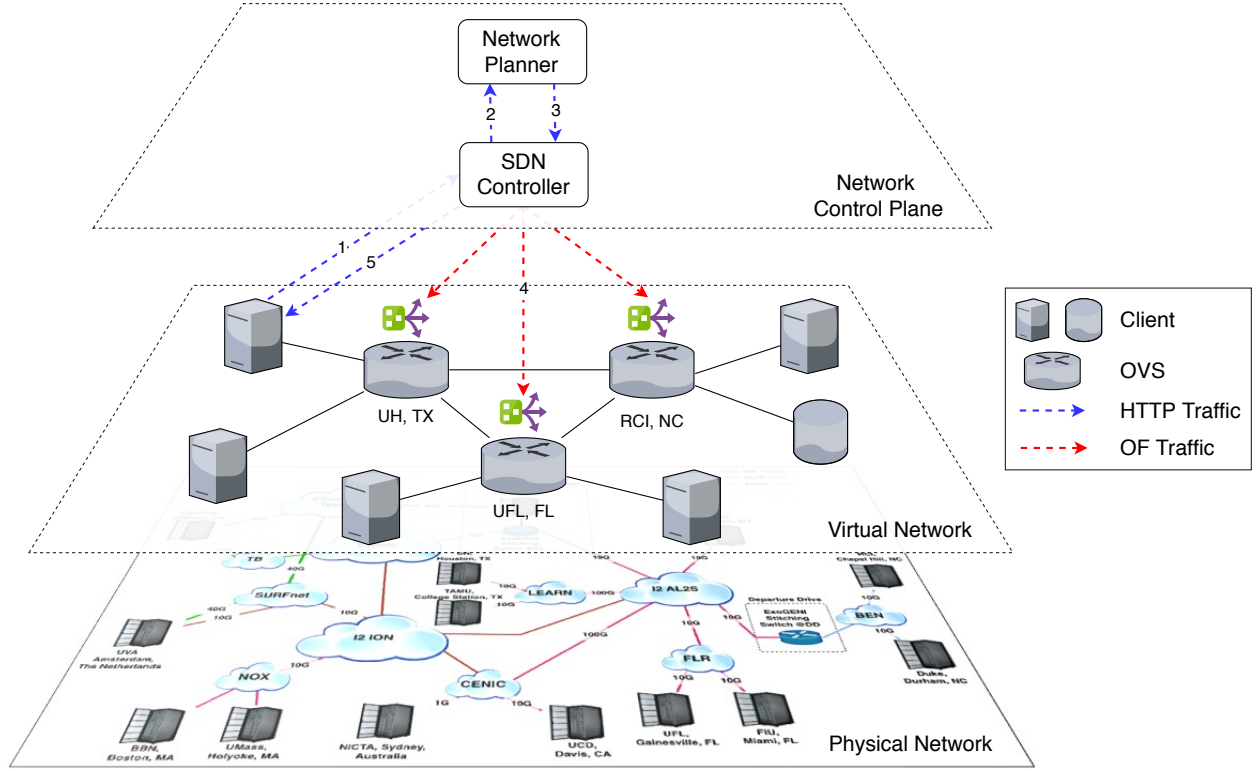


Figure 3.4: Network architecture in a *collaborative infrastructure*

3.3.2 Virtual Network

The *virtual network* is an abstraction of the network used in a *collaboration* for data transfers, which hides the complexities in the underlying *physical network* and provides a simple view of the network to ease the network management. Each *collaboration* has an instance of *virtual network* isolated from others, which only connects the resources reserved for the *collaboration*. Since we construct the *virtual network* as an SDN network, the control and data planes of the network are separated – the *network control plane* abstracts the control plane, while the *virtual network* serves as the data plane. A *virtual network* consists of *clients* and *switches*.

A *client* is a resource that can generate and receive data transfer traffic. To clarify the addressing, each *client* is uniquely addressed by a media access control (MAC) address and a private IP address. A daemon process runs on each *client* to monitor the network traffic and take network control directives to limit the sending rate of traffic through a REST API (Table 3.2). We implement the rate limiting using the hierarchical token bucket (HTB) algorithm in the Linux traffic control utility tc ⁹.

⁹ tc Linux manual page: <https://linux.die.net/man/8/tc>

A *switch* is an SDN-enabled network switch with only the data plane, which forwards network traffic according to the network flows installed on it. It is deployed at each geo-location where resources are provisioned for the *collaboration*, serving as a network gateway for the resources at each geo-location. For instance, as shown in Figure 3.4, three *switches* are deployed at Texas, Florida, and North Carolina in proximity to the resources provisioned for the *collaboration*. The *switch* does not own any logic but takes directives from the *network control plane* to route network traffic. In our implementation, we follow the norm in the SDN implementation by using OpenFlow (OF) (McKeown et al., 2008) as the communication protocol between the *network control plane* and the *switches*. Besides, since OF-compatible switches are rare, we implement the *switch* using the Open vSwitch (OVS) (Pfaff et al., 2015) instead, which is a kernel-based, OF-compatible network switch running on commodity servers. The use of OVS greatly improves the flexibility of *virtual network* deployment, as the *switches* can be deployed on any baremetal or VMs, which are widely available across many locations and platforms. Optionally, the *switches* are also responsible for bandwidth probing if bandwidth provisioning is not guaranteed and variable in the network. We employ a non-intrusive bandwidth estimation tool named Assolo (Goldoni et al., 2009) to probe network bandwidth with minimal bandwidth wastage and acceptable accuracy. The *switches* perform the bandwidth probing periodically and report the results along with other metrics, e.g., network latency, packet loss rate, to the *network control plane* to drive the network routing algorithm.

3.3.3 Network Control Plane

The *network control plane* controls the network traffic flows in the *virtual network* underneath. It runs a custom network routing algorithm to create network flows and send network control directives to install them on the *switches*. The *network control plane* owns a global view of the *virtual network*, kept updated by the *switches* of vital information about the *virtual network*, such as network topology, latency, and bandwidth utilization, which are partially or wholly used as the input parameters of the network routing algorithm. The *network control plane* is composed of the *network planner* and the *SDN controller*.

The *network planner* is an extensible, stateless service that runs network routing algorithms. It takes requests with a comprehensive snapshot of the *virtual network* from the *SDN controller* and generates *plans* as the algorithmic output, each of which is a set of network flows and bandwidth allocations to be executed on the *virtual network* (Figure 3.5(b)). The *network planner* accepts the installation of new algorithms through the REST API (Table 3.2). As the *SDN controller* provides only the abstract network state, the

algorithm implementation is decoupled from any specific network implementation. To install a new algorithm, the *network planner* captures its binary in the payload of the API call and convert it into an executable of the algorithm. The algorithm implementation must comply with the input and output schema, such that it can accurately capture the state of the *virtual network* and create results executable by the *SDN controller*. Figure 3.5 shows the examples of input and output of the *network planner*.

The *SDN controller* is the mediator between the *network planner* and the *virtual network*. It has a *northbound* and a *southbound* API to communicate with the *network planner* and the *virtual network*, respectively. Through the northbound API, the *SDN controller* submits the latest state of the *virtual network* to the *network planner* and receives the algorithmic result from it; through the southbound API, the *SDN controller* collects the network metrics from the *virtual network* and sends directives to control network flows. Internally, the SDN controller filters and organizes the network metrics into the state compliant with the input schema of the *network planner*, and translates the algorithmic results into clear directives executable on the *virtual network*. To align with the *virtual network*, we implement the *SDN controller* as an OF controller using the RYU framework¹⁰. The network control directives are encoded in OF flows, which are installed in the flow table of OVS to forward packets accordingly. With the OF 1.3 used in RADII, bandwidth limit can be set on an OF flow, which enables flow-level bandwidth allocation and thus sophisticate network routing algorithms can be applied in the *virtual network*. The algorithm we will introduce in Section 3.4 partly depends on this feature.

3.3.4 Data Transfer Mechanism

Here we mainly focus on TCP data transfers as TCP is dominantly used for remote bulk data transfers. In a *collaboration*, a data transfer is initiated from one *client* to another in the *virtual network*. We refer to the *client* initiating the transfer as *source* and the one accepting the transfer as *destination*. A TCP data transfer takes place on a TCP connection established between the *source* and *destination*, which is uniquely identified, as shown in Figure 3.5(a), by a quadruple of *source* and *destination* IP addresses and port numbers. To perform a data transfer, the *virtual network* and *network control plane* need to interact with each other to reliably build the connection and route packets between the resources.

¹⁰RYU framework: <https://osrg.github.io/ryu/>

Service	Endpoint	Method	Description
Client	/limit	PUT	Set a rate limit for a transfer
	/limit/<id>	POST	Update the rate limit of transfer <id>
		DELETE	Lift the rate limit of transfer <id>
SDN Controller	/transfer	PUT	Add a transfer
	/transfer/<id>	DELETE	Delete a transfer
	/transfer/<id>/meta	GET	List the metadata entries of a transfer <id>
		POST	Associate a metadata key-value entry to a transfer <id>
	/transfer/<id>/meta/<key>	DELETE	Delete a metadata entry of the transfer <id> by <key>
	/metrics	POST	Report network metrics
Network Planner	/algorithm	GET	List the supported network routing algorithms and their specifications
		PUT	Install a new network routing algorithm in binary format in the payload
	/algorithm/<id>	POST	Invoke the network routing algorithm <id>

Table 3.2: REST API of key components in the network of *collaborative infrastructure*

Figure 3.4 illustrates the interactions between the *virtual network* and the *virtual network*. A data transfer starts from a *client* notifying the *SDN controller* of the new transfer – it creates an instance of data transfer in the *SDN controller* through its API (Step 1). The *SDN controller* then initiates a request to the *network planner*, which contains all the ongoing data transfers and the latest state of the network (Figure 3.5(a)), to invoke the network routing algorithm (Step 2). Once the algorithm finishes, the *network planner* sends the result (Figure 3.5(b)) in the response back to the *SDN controller* (Step 3). The *SDN controller* converts the algorithmic results into network control directives, i.e. OF flows, and populates them into the flow tables of the *switches*, i.e., OVSs, such that packets of the data transfer will be forwarded along the assigned network path at a rate capped by the allocated bandwidth (Step 4). Lastly, the *SDN controller* sends an ACK to the *source*, indicating whether the network flow for the data transfer is successfully deployed and its bandwidth allocation (Step 5).

Despite the per-flow bandwidth limit in the *switches*, the *source* also limits its sending rate to comply with the bandwidth allocation made by the *network control plane*. The goal of the bandwidth throttling at the client side is to avoid unnecessary packet loss, since excessive packets tend to overflow the buffer of *switches* along the path and force packet loss, which is detrimental to a TCP data transfer. Due to the AIMD congestion control mechanism, TCP is sensitive to packet loss and drastically shrink its congestion window

```

{
  "transfers": [
    ...
    {
      "src": "10.0.0.1",
      "src_port": 10000,
      "dst": "10.0.0.2",
      "dst_port": 10001,
      "metadata": [
        {
          "key": "bw_demand",
          "value": 300
        },
        {
          "key": "priority",
          "value": 1
        }
      ]
    }
    ...
  ],
  "network": [
    ...
    {
      "from": "s1",
      "to": "s2",
      "bandwidth": 100.5,
      "latency": 20.3,
      "packet_loss": 0.1
    }
    ...
  ]
}

```

(a) input

```

{
  "flows": [
    {
      "src": "10.0.0.1",
      "src_port": 10000,
      "dst": "10.0.0.2",
      "dst_port": 10001,
      "path": ["s1", "s2", "s3"],
      "bw": 50.0
    }
    ...
  ]
}

```

(b) output

Figure 3.5: Examples of input and output of the *network planner*

to reduce throughput upon packet loss. Consequently, it leads to bandwidth underutilization even though the bandwidth is pre-allocated to the data transfer. This issue has a profound impact on the performance of remote TCP data transfers on bandwidth-reversed networks. As we will discuss in Section 3.5, we have developed a sender-side TCP enhancement to alleviate the problem.

It is also worth noting that bandwidth allocation is optional for network routing algorithms adopted in the *network control plane*. The SDN network barely sets any limitations on the selection of the network routing algorithm, and can adapt to conventional algorithms designed for the traditional network architecture, e.g., equal-cost multi-path (ECMP) (Hopps, 2000). Nevertheless, in the next section, we will introduce a network optimization solution built on top of bandwidth allocation.

3.4 Priority-Based Global Network Optimization

The SDN network enables network management in *collaborations* and offers the flexibility to adopt custom network management algorithms. However, the poor efficiency of data sharing in *collaborations* has yet to be addressed.

The root cause of inefficient data transfers in *collaborations* is the potentially severe network congestion. Since geo-distributed resources in a *collaboration* mostly communicates over WANs wherein bandwidth is relatively constrained, network paths can be easily saturated and thus network congestion occurs. More commonly, without a global view of the network, traditional routing algorithms tend to greedily allocate network paths to network traffic, creating congestion on certain paths while leaving others idle. The suboptimal resource allocation model leads to serious bandwidth wastage and thus aggravates the network congestion.

With the abstraction of the network control plane, SDN allows the routing algorithm to acquire comprehensive metrics of the entire network (as shown in Section 3.3) and perform global network optimization in a centralized manner. Hence, we propose a linear program (LP) extended from (Hong et al., 2013) to optimize bandwidth allocation and eliminate network congestion.

The original LP by Hong et al. (2013) is a multi-commodity flow (MCF) function at its core as follows:

$$\max \quad \sum_i b_i - \epsilon \left(\sum_{i,j} w_j \cdot b_{i,j} \right)$$

$$\text{s.t.} \quad b_{Low} \leq b_i \leq \min\{d_i, b_{High}\}, \quad \forall i \notin F \quad (3.1)$$

$$b_i = f_i, \quad \forall i \in F \quad (3.2)$$

$$\sum_{i,j} b_{i,j} \cdot I_{j,l} \leq c_l^{remain}, \quad \forall l \quad (3.3)$$

$$b_{i,j} \geq 0, \quad \forall (i, j) \quad (3.4)$$

In this function, i and j denotes a data transfer between a pair of *source* and *destination* and a network path between them, respectively. F is a set of data transfers with bandwidth allocated before the function invocation, and f_i represents the bandwidth allocation to each pre-existing data transfer i in F . c_l represents the capacity of a network link l and $I_{j,l}$ indicates whether a network path j uses the network link l . d_i and b_i

stand for the bandwidth demand of and allocation to a new data transfer i . Since a data transfer can consist of multiple network flows by using multipathing techniques, e.g, MPTCP (Raiciu et al., 2012), bandwidth allocation to a data transfer i is a sum of allocations to its network flows assigned to every network path j , i.e., $b_i = \sum_j b_{i,j}$. w_j is the latency of a network path j and ϵ is a small constant.

The MCF is a multi-objective LP, maximizing the total bandwidth allocation to data transfers in the network while minimizing the total latency of network paths being selected. In other words, the MCF tends to assign a data transfer to a network path with lower latency if its bandwidth demand can be satisfied. To guarantee the fairness among data transfers, an upper and lower bound of bandwidth allocation to a data transfer, b_{Low} and b_{High} , are pre-calculated by a max-min fairness function (Nace et al., 2006). The Constraint 3.1 specifies that, for any data transfer i without bandwidth allocation, its bandwidth allocation must fall between the max-min fairness bounds; if its demand d_i is less than the upper bound b_{High} , its allocation should be capped by its demand. The Constraint 3.2 indicates that the bandwidth allocation to pre-existing data transfers should remain unchanged. The Constraint 3.3 is a capacity constraint that ensures the total bandwidth allocation on a network link l will not exceed the remaining bandwidth available on it. Constraint 3.4 guarantees that bandwidth allocation to any network flow of any data transfer, i.e., $b_{i,j}$, must be non-negative.

By solving the MCF iteratively, this approach can reach a global optimum of bandwidth allocation to data transfers in the network, at which bandwidth utilization is maximized and network flows of data transfers are coordinated such that network congestion is mostly prevented, as proven by Hong et al. (2013).

The basis of the MCF is the assumption that bandwidth demand of each data transfer is clearly known, which is valid for services with defined service-level agreements (SLAs) – they are required to meet the minimum throughput requirement imposed in the SLAs. However, this assumption can barely hold in the context of *collaboration*, wherein data transfers are mostly spontaneous and the bandwidth demands are unclear. Instead, data transfers may have different priority – data transfers of time-critical data processing jobs have higher priority over data backup workloads. The priority can be quantified by levels; for instance, Level 9 stands for the highest priority, while Level 1 means the lowest one. The level of priority can be embedded in the metadata of a data transfer and used as an evidence for the network optimization, as introduced in Section 3.3.

Hence, we have extended the MCF proposed by Hong et al. (2013) with priority levels of data transfers. The extended MCF, referred to as *MCF+*, is formulated as follows:

$$\begin{aligned}
\max \quad & \sum_i b_i - \epsilon (\sum_{i,j} w_j \cdot b_{i,j}) \\
\text{s.t.} \quad & \frac{b_i}{p_i} = \frac{b_{i'}}{p_{i'}}, \quad \forall i, i' \notin F \quad (3.5) \\
& b_i = f_i, \quad \forall i \in F \quad (3.6) \\
& \sum_{i,j} b_{i,j} \cdot I_{j,l} \leq c_l^{remain}, \quad \forall l \quad (3.7) \\
& b_{i,j} \geq 0, \quad \forall (i,j) \quad (3.8)
\end{aligned}$$

We use p_i to denote the priority level of a data transfer i . Data transfers with greater value of p_i tend to have a higher priority level. The extension is made at the Constraint 3.5. Instead of allocating bandwidth as per demand, the *MCF+* allocates the bandwidth to data transfers in proportion to their priority levels. As a result, for new data transfers, higher-priority data transfers will acquire more bandwidth than lower-priority ones proportionally to their priority levels. The rest of this function remains the same as the original MCF.

However, the *MCF+* may cause starvation – low-priority data transfers may hog the bandwidth and starve high-priority ones if they enter the network earlier. The root cause is that the bandwidth allocation is uncapped but only proportional to the priority levels of data transfers. For instance, when low-priority data transfers are initiated in an idle network, they have the relatively highest priority in the network and will fully utilize the network bandwidth on the assigned network paths. Due to the Constraint 3.6, when a high-priority data transfer entering the network, it cannot acquire the bandwidth allocation in the amount proportional to its priority level, since the constraint guarantees the pre-existing bandwidth allocation remains unchanged.

To address this issue, we cap the bandwidth allocation to pre-existing data transfers using the following formula in prior to invoking the *MCF+*:

$$b_i = \max\{b_i, \frac{p_i}{\sum_{i' \in F'} p_{i'}} \cdot \sum_{i'' \in F} b_{i''}\} \quad (3.9)$$

Here we use F' to denote the set of network flows of both pre-existing and new data transfers, i.e., $F' = \{i | i \in F\} \cup \{i | i \notin F\}$. The formula hypothetically reallocate the previously allocated bandwidth

among all the data transfers, such that the pre-existing data transfers will spare certain amount of bandwidth for the new ones in proportion to their priority levels. Thus, bandwidth allocation to pre-existing data transfers is capped to their fair share regarding their priority levels and new data transfers will never be starved.

The *MCF+* optimizes the network based on the priority of data transfers, maximizing the bandwidth utilization and avoiding network congestion in the network. From the perspective of data transfers, they no longer need to contend for bandwidth with others but acquire a fair share of bandwidth allocation, thus resulting higher and more stable throughput on average. We will show the evaluation of the *MCF+* in Section 3.6.

3.5 TCP Mario For Bandwidth-Reservable Networks

With *MCF+*, data transfers are designated to optimal network paths and allocated with a fair amount of bandwidth. The bandwidth allocation can be guaranteed in bandwidth-reservable networks, in which dedicated network connections are established with guaranteed bandwidth provisioning, such as ExoGENI, AWS Direct Connect¹¹, GCP Dedicated Interconnect¹², among others.

However, TCP data transfers in such networks inherit the classic congestion control mechanism, which probes the maximum fair throughput in an AIMD manner. The throughput can be estimated using the equation as follows:

$$throughput \leq \frac{C \cdot MSS \cdot cwnd}{RTT} \quad (3.10)$$

In this equation, C denotes a constant. As illustrated, the throughput is positively correlated with the MSS and the size of the congestion window denoted by $cwnd$, and negatively with the RTT between the two ends of the TCP connection.

The equation is derived based on the behavior of the TCP congestion control mechanism – the sender increases the $cwnd$ in an additive manner upon the receipt of every acknowledgement (ACK), which takes an RTT to travel from the receiver to the sender; therefore, the sender takes at least one RTT to expand the $cwnd$ and reach the maximum size. Additionally, the TCP congestion control recognizes packet loss as the

¹¹AWS Direct Connect: <https://aws.amazon.com/directconnect>

¹²GCP Dedicated Interconnect: <https://cloud.google.com/interconnect/docs/concepts/dedicated-overview>

signal of network congestion – it reduces the *cwnd* in a multiplicative manner upon packet loss to throttle the throughput to a fair rate.

With bandwidth pre-allocated, bandwidth probing is no longer necessary for data transfers; instead, they should send traffic at a rate that fully utilizes the allocated bandwidth. However, the throughput of TCP data transfers is still constrained by the RTT and random packet loss due to the congestion control mechanism, which is unrelated to the throughput in a bandwidth-reservable network. Herein, we devise the TCP MARIO to decouple TCP data transfers from the traditional congestion control mechanism and maximize the throughput using pre-allocated bandwidth.

Let b denote the bandwidth allocation to a data transfer and replace the throughput with it in the Mathis equation, we can derive the following equation:

$$cwnd \geq \frac{b \cdot RTT}{C \cdot MSS} \quad (3.11)$$

TCP MARIO uses this equation to calculate the *cwnd* on the sender side during the establishment of the TCP connection. Thus, it skips the bandwidth probing phase but reaches the maximum throughput instantly. Further, since bandwidth is pre-allocated and guaranteed, TCP MARIO assumes any packet loss is a random loss and will never back off the *cwnd* upon packet loss. It only readjusts the *cwnd* when the bandwidth allocation changes.

In the implementation, TCP MARIO is developed as a sender-side TCP congestion control module in the Linux kernel¹³. It is installed on the *client* in the *virtual network*, which tends to initiate data transfers. The kernel module learns of the bandwidth allocation to data transfers from the daemon process running on each *client*, which updates the bandwidth allocation based on the response from the *network control plane*. In Section 3.6, we compare TCP MARIO to other baseline TCP variants to evaluate its performance in data transfers.

3.6 Experimental Evaluation

In this section, we present the experimental evaluation on RADII as a full-fledged geo-distributed system for performing data-centric collaborations. Specifically, we have run bulk data transfers on a geo-distributed deployment to evaluate the effectiveness of the global network optimization, *MCF+*, and the

¹³TCP MARIO: <https://github.com/dcvan24/tcp-mario>

TCP enhancement, TCP MARIO, proposed in Section 3.4 and 3.5, respectively. First of all, we introduce the experiment setup.

3.6.1 Experiment Setup

The experiment is conducted on an instance of RADII deployed across four geo-locations in ExoGENI as shown in Figure 3.6, which are UH (Houston, TX), UFL (Gainesville, FL), UMass (Amherst, MA), and PSC (Pittsburgh, PA). At each location, we have deployed an iRODS resource node on a VM with 4 CPU cores, 12GB RAM and 75GB disk space, which serves as a *client* sending and receiving files over the WAN. The *virtual network* is built on top of four *switches* powered by Open vSwitch, each of which runs at a geo-location on a VM with 2 CPU cores, 6GB RAM, and 50GB disk space. The virtual switches are interconnected with 600 megabits per second virtual network links, on which the bandwidth provisioning is guaranteed. The bandwidth between the *client* and *switch* is 1.8 gigabits per second. The *network control plane* is deployed at SL (Chicago, IL), in which the *SDN controller* and *network planner* run separately on VMs with 4 CPU cores, 12GB RAM, and 75GB disk space. The *SDN controller* and *network planner* are implemented using RYU SDN framework and SageMath¹⁴, respectively. The middleware of RADII is mainly developed using Python 2.7¹⁵.

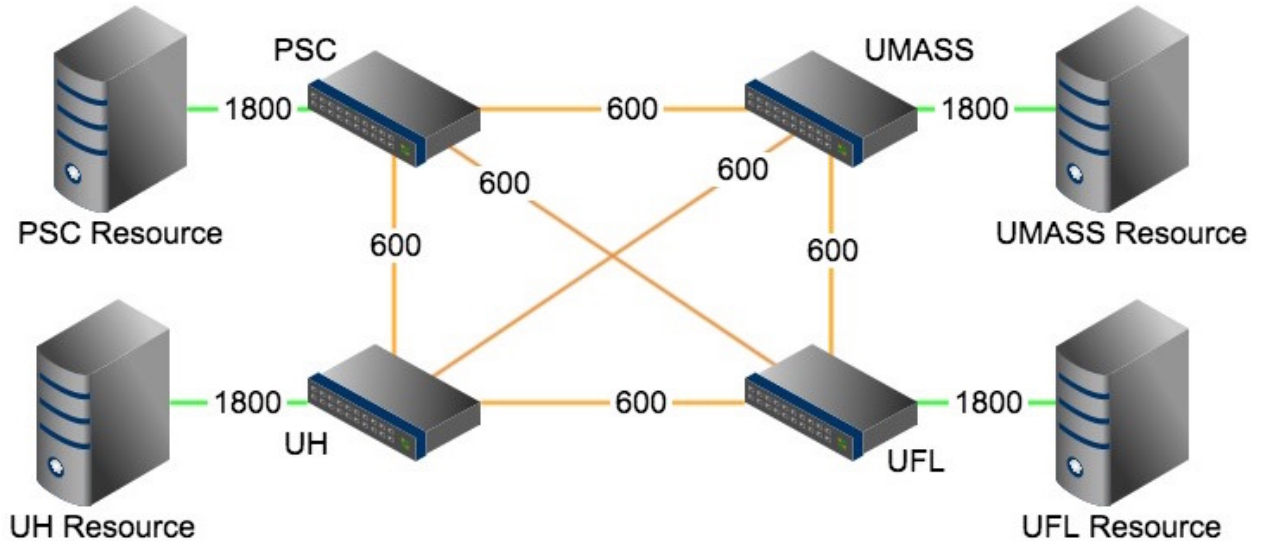


Figure 3.6: Network topology for evaluation

¹⁴SageMath:<http://www.sagemath.org>

¹⁵Python 2.7.0 Release: <https://www.python.org/download/releases/2.7>

To evaluate the *MCF+*, we use the ECMP as the baseline algorithm, which is representative of traditional network routing algorithms adopting the distributed resource allocation model. The experimental workload consists of data transfers of file in three different sizes: 256MB, 1GB, and 4GB. The priority level of a data transfer is inversely correlated with the file size – a data transfer of a smaller file has a higher priority level. The intuition of this workload design is that transfers of smaller files, e.g., ad-hoc files transfers, usually have higher urgency than those of larger ones, e.g., daily data backup, which is aligned with the real-world workloads at Microsoft (Hong et al., 2013). Specifically, we assign priority levels of 3, 5, and 7 to data transfers of 4GB, 1GB, and 256MB files, respectively. For TCP congestion control, we use TCP CUBIC on both ends of each data transfer. We use throughput of data transfers and overall bandwidth utilization as the metrics to evaluate the effectiveness of the *MCF+*.

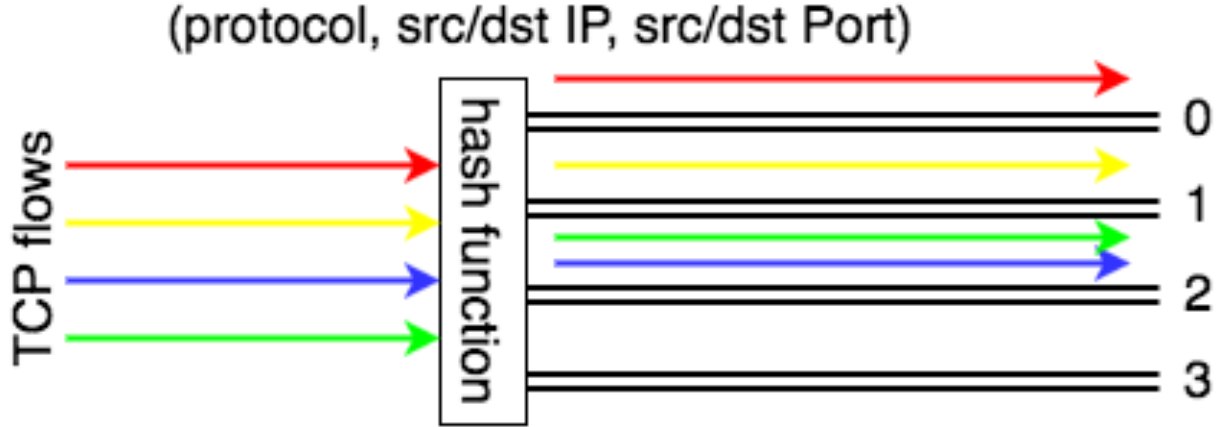


Figure 3.7: The ECMP algorithm

To evaluate TCP MARIO, we select the TCP congestion control algorithms listed in Table 3.3 as baselines, which have also been briefly introduced in Section 2. In the experiment, we compare TCP MARIO to the baselines in network conditions with varying bandwidth (100Mbps–5Gbps), latency (0–100ms), and packet loss rate (0-0.05%). We use NetEm (Hemminger, 2005) to emulate the varying network conditions. We use two workloads to drive the experiment and investigate the different aspects of TCP MARIO. In the first workload, we use *iperf3*¹⁶ to generate and transmit synthetic data over the network for 60 seconds, which emulates regular bulk data transfers from 750MB to 37.5GB. In this experiment, we focus on the bandwidth utilization of data transfers resulted by different algorithms.

¹⁶*iperf3*: <http://software.es.net/iperf/>

Algorithm	Description
Reno (Jacobson, 1988)	An implementation with fast recovery and fast retransmission
Veno (Fu and Liew, 2003)	A Reno extension for wireless network
Vegas (Brakmo and Peterson, 1995)	A Reno extension with proactive congestion estimation
Westwood (Mascolo et al., 2001)	A TCP variant that estimates bandwidth
H-TCP (Leith and Shorten, 2004)	An implementation that aggressively utilizes bandwidth on network paths with large BDP
CUBIC (Ha et al., 2008)	An implementation that utilizes available bandwidth in real time

Table 3.3: Baseline TCP congestion control algorithms

3.6.2 Evaluation of *MCF+*

First, we evaluate the *MCF+*. Figure 3.8 shows the comparison of data transfer throughput between the ECMP and *MCF+*. As shown, the *MCF+* outperforms ECMP in throughput for data transfers with any priority level. The result is expected since the *MCF+* optimizes the path assignment for data transfers to avoid network congestion; in contrast, the ECMP distributes network flows among network paths in a non-deterministic fashion (as shown in Figure 3.7), likely to cause network congestion and thus affect the throughput of data transfers. The non-deterministic path assignment of the ECMP also affects the stability of data transfer throughput. As reflected in Figure 3.7, the ECMP results larger variation of throughput among data transfers than the *MCF+*. Figure 3.9 further clarifies the instability by showing the throughput variation of 50 data transfers between UFL and UMass resulted by the ECMP and *MCF+*, respectively. The observation is intuitive as the throughput will be high when the data transfers are evenly distributed without collisions, but low when they are packed onto only a few network paths where network congestion occurs. In comparison, the *MCF+* always distributes data transfers across network paths and prefers those with smaller network latency, and thus the data transfer throughput is more stable and predictable.

The optimal bandwidth allocation performed by the *MCF+* also contributes to the higher throughput. The bandwidth allocation is enforced in the *virtual network* by rate limiting, such that data transfers can only utilize their allocated bandwidth but rarely contend with each other. In comparison, the ECMP does not perform bandwidth allocation but allows data transfers to send traffic into the network indulgently. Consequently, the excessive network traffic aggravates the network congestion and further harms the throughput of data transfers.

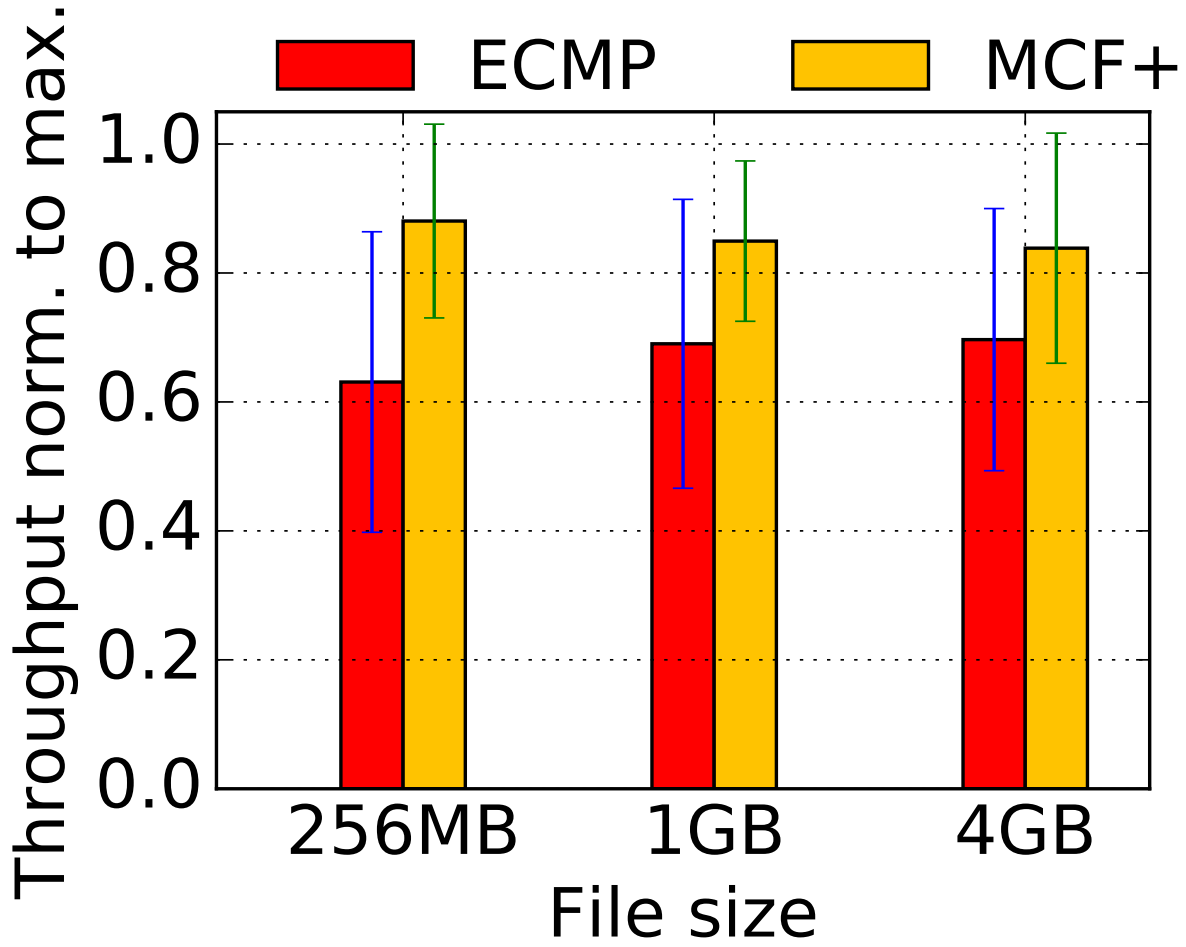


Figure 3.8: Comparison of average data transfer throughput between ECMP and MCF+

Besides, as illustrated in Figure 3.8, the *MCF+* favors data transfers of smaller files (i.e., higher-priority) to attain higher throughput than those of larger ones (i.e., lower-priority). This behavior is consistent with the design of the *MCF+* in overall, although the throughput is not strictly proportional to the priority levels as expected. The intuition behind this is that these TCP data transfers of small files have shorter duration and thus are more vulnerable to random packet loss. The random packet loss falsely forces the TCP congestion control to reduce the throughput, which may not recover till the end of the data transfers. Consequently, they fail to fully utilize the allocated bandwidth due to the unexpected slowdown and thus result low throughput. In contrast, data transfers of larger files have sufficient time to recover from the slowdown and stay at a high level of bandwidth utilization. Herein, the slowdown caused by random packet loss leads to the disproportionate throughput of data transfers as compared to their priority levels.

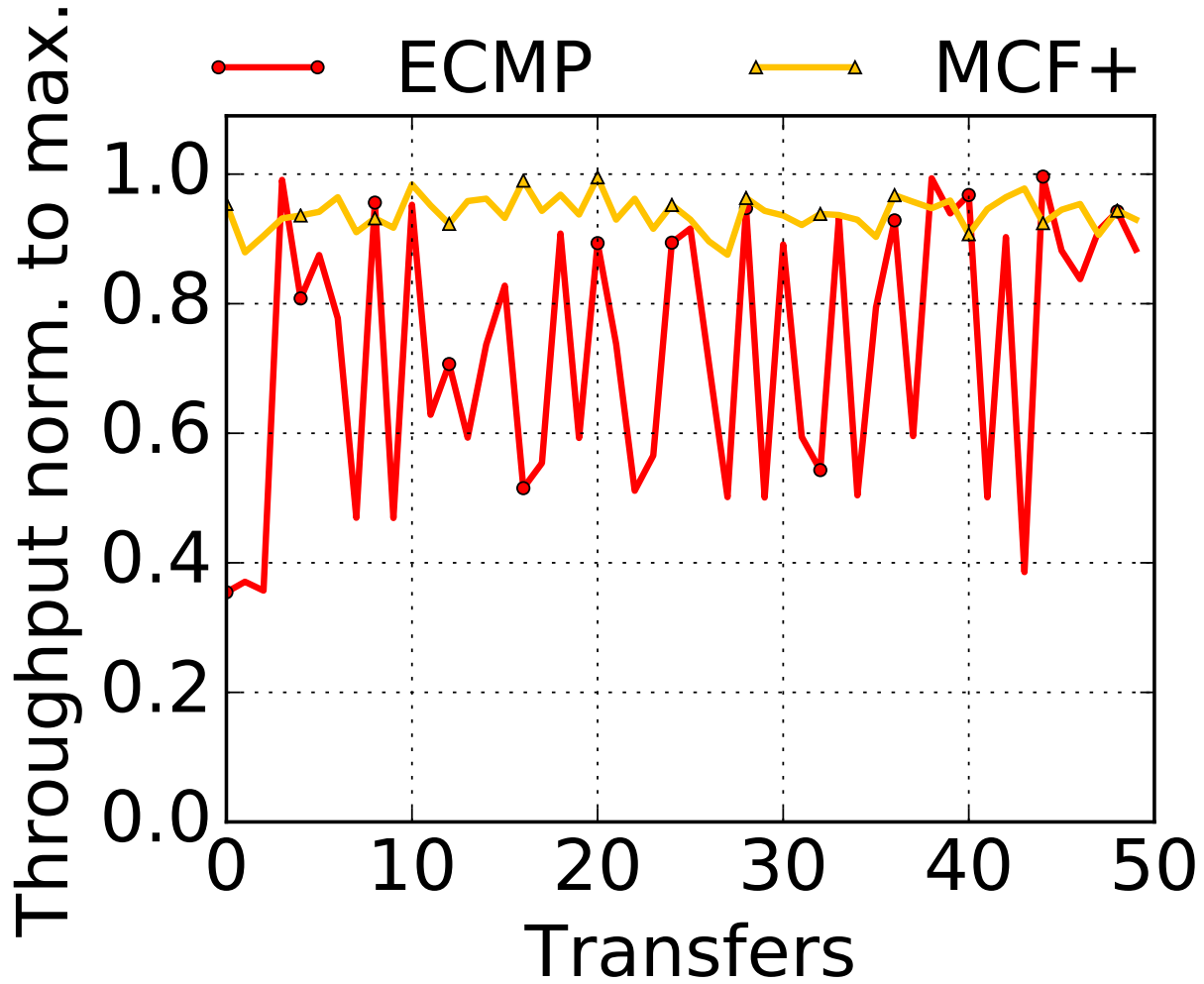


Figure 3.9: Throughput variation of 50 data transfers between UFL and UMASS

We also evaluate the *MCF+* from the system perspective by investigating the overall bandwidth utilization as illustrated in Figure 3.10. We can observe that the *MCF+* defeat the ECMP in overall bandwidth utilization by roughly 25% on average because of the optimization of bandwidth allocation and path assignment for data transfers. Moreover, the bandwidth utilization of the *MCF+* is more consistent than the ECMP during the observation period. The experimental results reveal that RADII benefits from the global view of network enabled by the SDN network to improve the resource utilization in the system.

3.6.3 Evaluation of TCP MARIO

Next, we evaluate TCP MARIO. Figure 3.11 shows the bandwidth utilization of TCP MARIO and the baselines in varying network conditions. In overall, we observe that TCP MARIO is more resilient to network

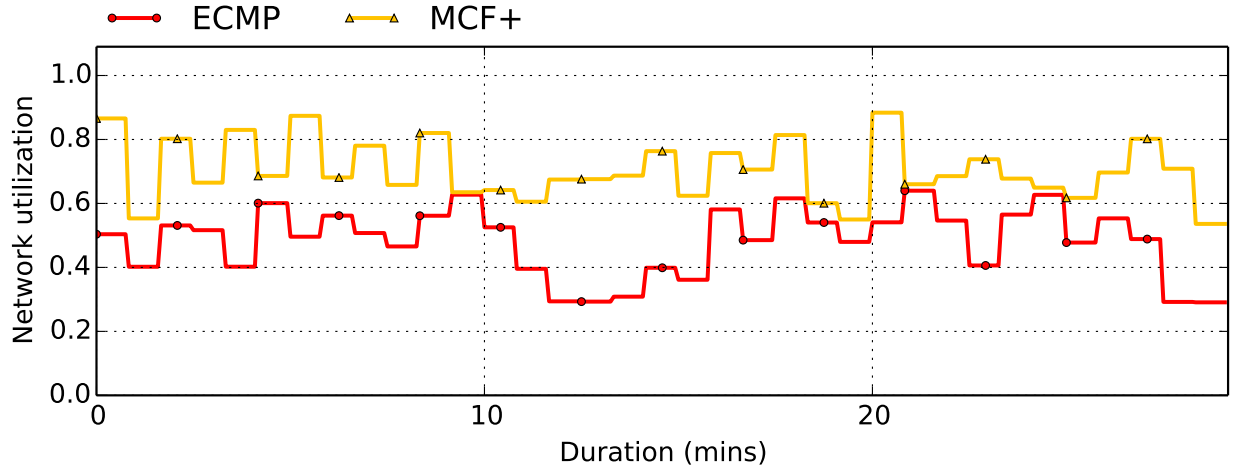


Figure 3.10: Variation of overall bandwidth utilization in 30 minutes

latency and packet loss than the baselines, especially with large bandwidth allocation (e.g., 5 gigabits per second). Specifically, in the network condition with minimal packet loss (leftmost column in the figure), the bandwidth utilization of the baselines is still affected by the increasing BDP as the latency increases – the bandwidth utilization of the baselines drops under 40% when the network latency is 40ms. As expected, the bandwidth utilization of TCP MARIO merely drops by 10% but constantly stays above 70% as the network latency increases. This is because TCP MARIO uses explicit bandwidth allocation instead of waiting on ACKs to estimate congestion window, and thus no longer subject to the Mathis’s equation, i.e., the throughput of TCP MARIO is not correlated with the network latency. Hence, the results demonstrate that TCP MARIO is able to achieve high throughput for remote data transfers over WANs.

In addition, as reflected in Figure 3.11, TCP MARIO is more tolerable to packet loss than the baselines, especially in a WAN (bottom row in the figure) – the bandwidth utilization of the baselines decreases below 10%, while TCP MARIO keeps the bandwidth utilization above 50% as the packet loss rate increases to 0.05%. This follows the intuition since TCP MARIO has the knowledge of bandwidth allocation and thus does not falsely shrink the congestion window as the baselines do; hence, it retains high throughput with the packet loss rate increasing. However, the bandwidth utilization of TCP MARIO is still affected noticeably by packet loss when the bandwidth is large. The rationale behind is that packet loss forces retransmission of the current congestion window, of which the size is positively proportional to the bandwidth allocation; therefore, the overhead for retransmission is large with large bandwidth allocation, which lowers the throughput of TCP MARIO. Herein, it reveals that TCP MARIO is tolerable to packet loss during data transfers.

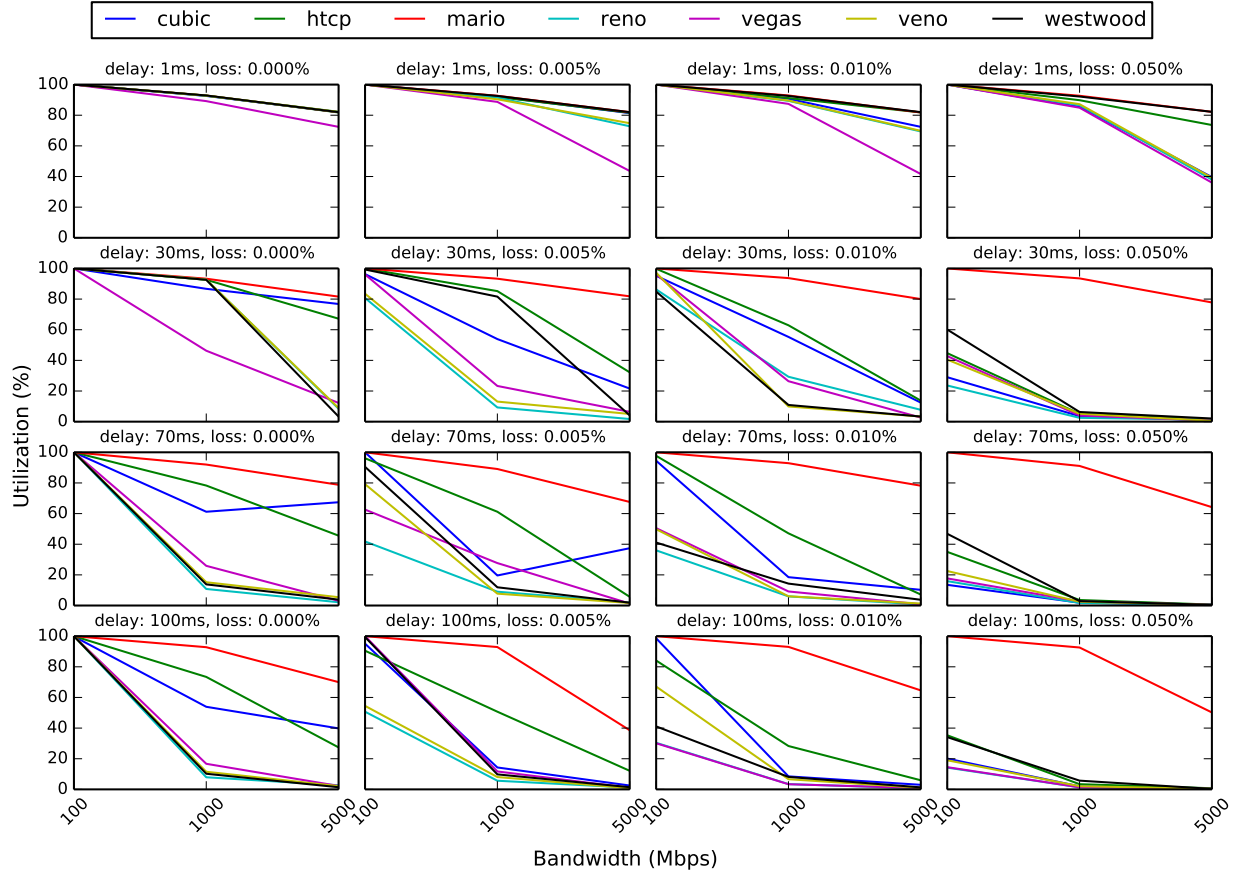


Figure 3.11: Comparison of bandwidth utilization among the selected TCP congestion control algorithms in different network conditions with varying bandwidth, latency and packet loss rate

3.7 Chapter Summary

In this chapter, we have introduced RADII, a full-fledged distributed system that enables scientific collaboration in the geo-distributed environment. We have proposed a DFD-based data model to abstract collaboration and computing infrastructure, bridging the gap between the high-level description of collaboration and low-level resource provisioning. Moreover, we have also recognized the inefficiency of data sharing in geo-distributed collaboration. As a solution, we have built an SDN-based virtual network for collaboration and devised MCF-based global network optimization and TCP MARIO on top, to improve the efficiency of data sharing in the geo-distributed environment. The experimental results demonstrate that RADII is able to support complex geo-distributed collaboration with efficient data sharing.

CHAPTER 4: Geo-Distributed, Network-Aware Caching for Data-Intensive Applications

In this chapter¹, we investigate caching strategies for improving the performance of data-intensive applications in geo-distributed environments and present our work CACHALOT in detail. As discussed in the prior chapter, we have recognized that scientific research of today is increasingly data-driven, collaborative and dependent on huge data sets that require geo-distributed computing and data sharing infrastructure. For data processing, scientists rely on a myriad of geo-distributed computing resources, such as high-performance computing (HPC) clusters, cloud-hosted data analytic services, and national CI. To use these resources, they move large volumes of data over the WAN.

However, performance overhead for data movement over the WAN is considered the root cause of poor performance of geo-distributed applications as recognized in JetStream (Rabkin et al., 2014), geo-distributed Spark (Pu et al., 2015), Geode (Vulimiri et al., 2015b), and other studies. Network-driven approaches, such as the SDN-based approach introduced in the prior chapter, have emerged as viable solutions to improve data transfer efficiency in these environments. Nevertheless, given the slowdown in network capacity expansion and the rapid growth of data volumes, adopting network-based approaches solely may not be sufficient to overcome the aforementioned challenges.

Alternatively, caching has the potential to alleviate the problem of massive data movement over WAN by trading off storage at the network edge for bandwidth saving at the network core. Specifically, they exploit cache space at the network edge to absorb network traffic by maintaining replicas of repeatedly used data objects near their frequent consumers. Furthermore, *computation caching* has been recently introduced to data processing frameworks, such as Spark (Zaharia et al., 2012), Nectar (Gunda et al., 2010), and Tachyon (Li et al., 2014), to avoid redundant re-execution of common computations, thus reducing network traffic and enabling efficient use of compute resources within the data center. Consequently, data-intensive applications perceive significant speedup and users observe noticeable time savings. *Cooperative caching* (Dahlin et al.,

¹Content of this chapter previous appeared in preliminary form in the following paper:
Jiang, F., Castillo, C., and Ahalt, S. (2018). Cachalot: A network-aware, cooperative cache network for geo-distributed, data-intensive applications. In NOMS 2018-2018 IEEE/IFIP Network Operations and Management Symposium. IEEE.

1994) extends the caching techniques to the WAN, which are proven effective in the Internet (Chand et al., 2007; Cao et al., 2007), mobile networks (Shafiq et al., 2014; Ming et al., 2014), and CDNs (Sourlas et al., 2013; Zhang et al., 2013). However, these approaches mainly optimize for network and disk I/O, but seldom consider the cost for generating new data sets not found in cache.

We develop CACHALOT, a novel network-aware, cooperative cache network that amortizes the cost of data transfers by means of caching output data sets of redundantly executed computation jobs; and, makes cache placement and replication decisions based on the availability of network resources. CACHALOT is utility-driven and builds on a combination of techniques that together perform intelligent trade-off decisions. CACHALOT retains in cache the data of highest user-defined utility value while simultaneously balancing the use of network and storage resources; and, improves completion time of data-intensive applications by taking into account the cost of executing jobs and generating data. To adapt to the ever-changing conditions of shared infrastructure, CACHALOT dynamically revises previous caching and replication decisions based on resource availability and data access history. Finally, CACHALOT builds on dynamic logical artifacts that enable the efficient design and development of caching algorithms.

For the rest of this chapter, we first formulate the problem formally (Section 4.1). Then, we describe the system design and its core network-aware cache algorithm (Section 4.2). Next, we present the results of a comparative evaluation of CACHALOT against several state-of-the-art baseline algorithms (Section 4.3). Finally, we summarize this chapter (Section 4.4).

4.1 Problem Formulation

We consider an environment where users, compute and data resources are geo-distributed over the WAN. Users submit jobs to compute resources. A job can be an instance of any kind of program specification, e.g., MapReduce (Dean and Ghemawat, 2008). Compute resources are hosted on premise or on public infrastructure, e.g., cloud platforms, and support big data processing frameworks, e.g., Spark. Similarly, data sets are available via public data repositories, e.g., National Center for Biotechnology Information (NCBI)², or on-premise databases. Users are located at institutions or sites with moderate storage resources for caching data products that may be reused in the future. We refer to these data sets or data objects interchangeably in this paper. A data object is *uniquely* identified by the computation job that produces it. Similarly, a job

²National Center of Biotechnology Information: <http://www.ncbi.nlm.nih.gov/>

is uniquely identified by information such as *executable binary*, *input data set* and *execution parameters*. The *utility* associated with a data object is not only a function of its potential demand or usage but also the performance improvement that it provides to users by virtue of its placement, and the impact on the corresponding network performance, as well as its size and the cost associated with generating it. Intuitively, caching a data object that can be quickly recomputed in a cache node with poor network connectivity does not bring much utility to future requesters.

To process data, users transfer input and output data sets over the WAN connecting compute and storage infrastructure. Thus, the end-to-end completion time of a single job consists of the time needed for processing and transferring data. Both, the sharing of oversubscribed compute resources and lossy and high-latency network paths result in unpredictable long execution times. By adopting an advanced caching strategy, data sets can be reused and transferred from near-by cache resources instead of recomputing jobs thus effectively reducing completion time and cost on behalf of the users.

We represent the cache system as a complete graph $G=(V, E)$, where V and E denote the set of cache nodes and network links between them, respectively. The capacity of each node $v \in V$ is denoted by C_v . The bandwidth and latency of each link $e \in E$ is denoted by b_e ($b_e > 0$) and l_e ($l_e \geq 0$), respectively. We use M to denote the set of data objects cached in G , in which size of each object $m \in M$ is denoted by s_m . We use $u_{m,v}$ to denote *utility* value of data object m on cache node v . The placement of data objects are represented by a binary matrix P : $P_{m,v}$ is 1 if data object m is placed in cache node v and 0 otherwise; $\sum_{v \in V} P_{m,v} = 0$ if a data object $m' \in M$ is no longer in cache. Since data objects can be transferred between cache nodes, we use binary matrix T to denote all possible data transfers: $T_{m,e}$ is 1 if data object m is transferred over link e and 0 otherwise. We aim at maximizing the total *utility* value of data objects in cache while minimizing cost for data transfers due to relocation and replication of data objects. The problem can be formulated as follows:

$$\begin{aligned}
\max \quad & \sum_{m \in M} \sum_{v \in V} P_{m,v} \cdot u_{m,v} - \sum_{m \in M} \sum_{e \in E} T_{m,e} \cdot \left(\frac{s_m}{b_e} + l_e \right) \\
\text{s.t.} \quad & \sum_{m \in M} P_{m,v} \cdot s_m \leq C_v & \forall v \in V \quad (4.1) \\
& \sum_{v \in V} P_{m,v} \geq 1 & \forall m \in M \quad (4.2) \\
& P_{m,v} \in \{0, 1\} & \forall m \in M, \forall v \in V \quad (4.3) \\
& T_{m,e} \in \{0, 1\} & \forall m \in M, \forall e \in E \quad (4.4)
\end{aligned}$$

Constraint (4.1) is the capacity constraint that ensures the total size of data objects cached in each node will not exceed the capacity of the node. Constraint (4.2) is the placement constraint that guarantees that every data object in cache has at least one copy available in certain cache nodes. Constraint (4.3) and (4.4) are binary constraints with P and T being binary matrices. We note that the binary constraints transform the problem into a 0-1 multiple knapsack problem (MKP), which is proven NP-complete (Kangasharju et al., 2002). We introduce an advanced network-aware caching algorithm to address this problem.

4.2 System Design

CACHALOT is a distributed, cooperative cache system that temporarily stores output data of executed jobs to avoid redundant job executions. It adopts an adaptive network-aware cache algorithm, which adjusts to ever-changing network conditions and exploits distributed cache capacity to reduce job completion time and better utilize resources. More importantly, CACHALOT is utility driven in that a *utility* value associated with data objects drives all cache activities including eviction, replication and placement of data objects. In CACHALOT, the utility captures the performance gain resulting from caching data objects based on their placement, usage and availability of network resources.

4.2.1 Architecture

CACHALOT is a cache network that consists of a *Cache Manager* (CM) and a number of interconnected *Cache Agents* (CAs) (Figure 4.1). Each CA is equipped with storage to cache data and runs an instance of a network-aware cache algorithm to manage its local cache. Its primary role is to serve as a first-level cache for

clients that are co-located with the CA, which are typically deployed in proximity to computing resources for running data-intensive computational jobs, such as HPC clusters, on-premise infrastructure, and cloud services. The CAs work cooperatively on data placement, retrieval, and replication decisions to reduce the completion time of jobs, while relying on the CM for bookkeeping of metadata of cached data sets such as location, size, access frequency, among others.

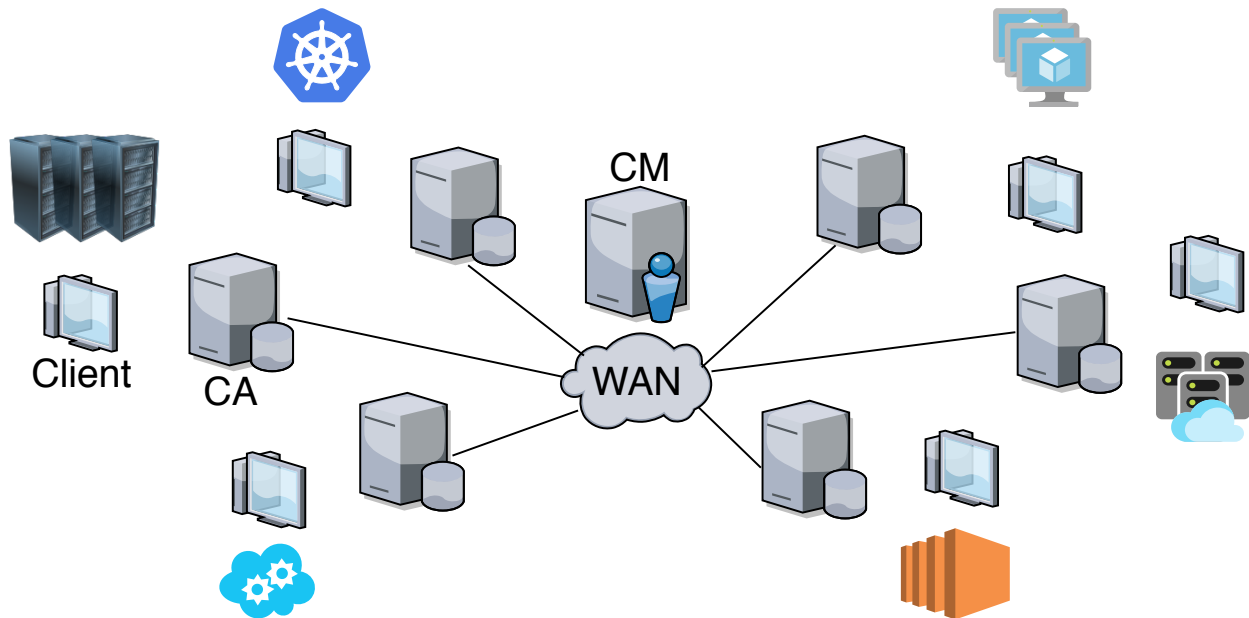


Figure 4.1: CACHALOT physical infrastructure

CACHALOT uniquely identifies a data set by the specification of the job producing it, which contains the vital information such as executable binary, input data set, parameters, among others. This information can be encrypted into a unique *fingerprint*, which is used as the digital object identifier (DOI) of the associated data set. Specifically, we use the SHA-1 algorithm (Eastlake and Jones, 2001) to generate a 160-bit DOI for each data set stored in CACHALOT. Other cryptographic algorithms, such as Rabin fingerprint (Broder, 1993) and message-digest algorithm 5 (MD5) (Rivest, 1992), can be used as the alternative for encoding data sets.

Figure 4.2 illustrates the interaction among the components in CACHALOT. Before executing a new job, the client contacts its local CA to check whether the output data set of the job exists in the local cache. If a replica of the data set exists, it is returned to the client immediately such that the full execution of the job is avoided (Step 2); otherwise, the local CA reaches the CM to find whether the target data set exists in any other remote CA (Step 3). If failed to locate the data set, the CM informs the local CA that initiates the request (Step 4), and the local CA then responds to the client that the output data set of the new job is not

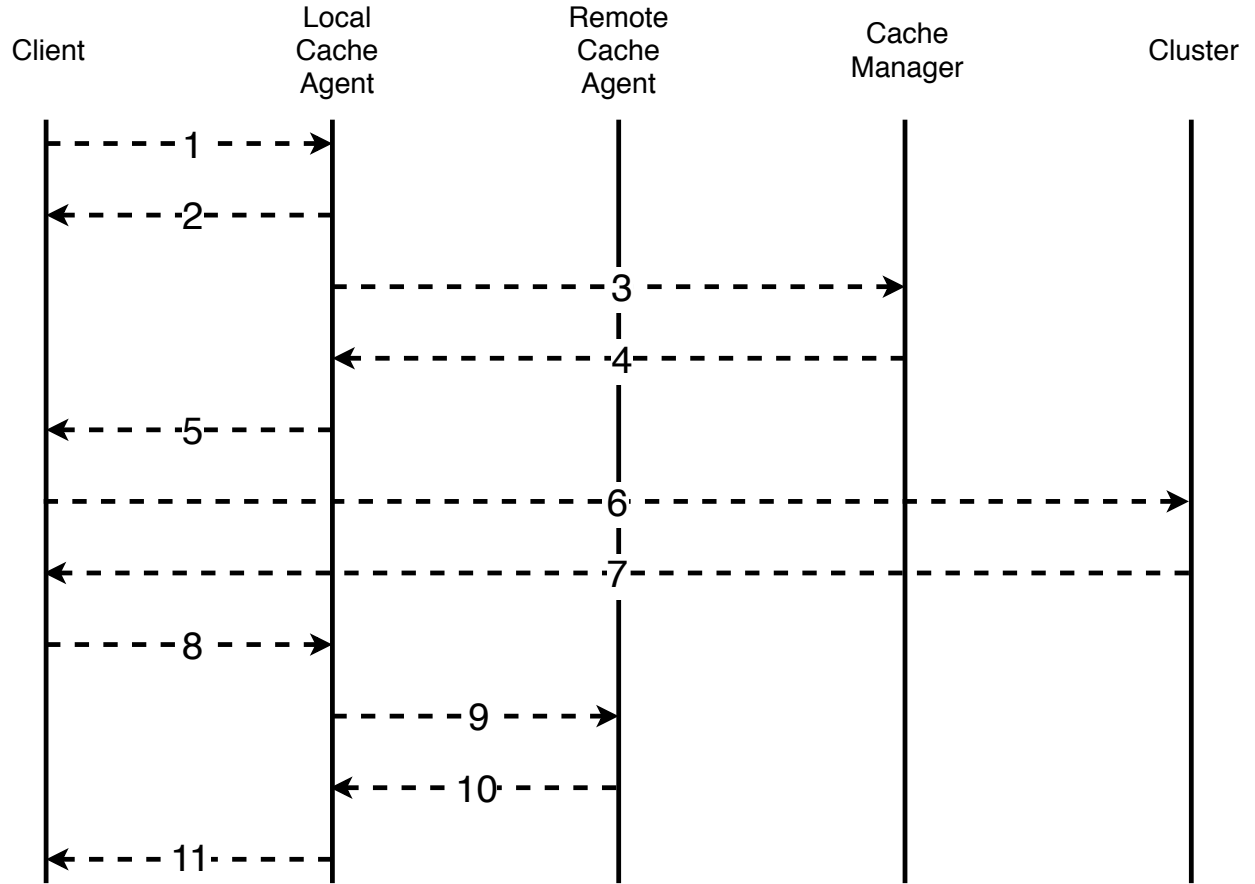


Figure 4.2: Interaction among the client, CA and CM in CACHALOT

cached (Step 5). In this case, the client sends the job to the local computing cluster for execution (Step 6) and receives the output data set once the job finished (Step 7). Then, the client voluntarily caches the output data set in CACHALOT for future reuse (Step 8). If the desired data set does exist in a remote CA, the CM returns the location of the remote CA, e.g., IP address, to the local CA (Step 3). The local CA directly requests for a replica of the data set from the remote CA and decides whether to replicate the data set locally (Step 10). Lastly, the local CA returns the output data set to the client (Step 11).

As CACHALOT running on top of geo-distributed CAs with cache space, it is critical to develop an efficient distributed algorithm to coordinate among the CAs and thus maximize the utilization of the geo-distributed cache. Furthermore, we realize that running the algorithm in a central manner creates a heavy load on the centralized server, e.g., CM, and consequently affects the system scalability. Further, the centralized algorithm execution can also cause single point of failure, compromising the system availability.

Hence, we distribute the execution of cache algorithm across CAs in CACHALOT. By design, each CA runs the cache algorithm independently of others – it reaches the CM only for the information about other CAs and the data sets of interest, which serves as the evidence for making algorithmic decisions. The cache algorithm running on the CA is customizable and pluggable – custom cache algorithms can be installed in CACHALOT in a *hot-plug* manner if they implement the API shown in Algorithm 1– 4. We have also developed a network-aware, distributed cache algorithm to improve the opportunity for finding desired data sets in cache. Before diving into the details of the algorithm, we first introduce the logical artifacts that support the cache algorithm.

4.2.2 On *datapods*, *dataserver* and *dataclients*

In CACHALOT, we refer to a data set in cache as a data object. Figure 4.3 illustrates the logical representation and artifacts introduced in CACHALOT for driving the network-aware cache algorithm. Naturally, clients retrieve replicas from the cache node that offers better network connectivity, i.e., bandwidth. Thus, CAs can be organized into logical clusters, each of which serves and manages one replica. Since replicas are created on demand, there can be multiple replicas for a given data object d at any point of time. We refer to such logical cluster as a *datapod*. There is a *datapod* for every replica. Each *datapod* consists of a *dataserver* and a group of *dataclients*, which are CAs at the backend. The *dataserver* hosts and serves the replica to its *dataclients*. The membership of a *datapod* consists of CAs that observe good network connectivity, e.g., bandwidth availability, with the *dataserver* node, as compared with other *dataserver* nodes hosting a replica of d . The CM maintains a catalog of cached data object records, each of which is mapped to a set of *datapods*. The record of a *datapod* specifies the address of the *dataserver*, a list of its *dataclients* and their retrieval history. As explained later, the CM also maintains network monitoring information to support decision-making processes and the membership of *datapods*. Intuitively, a *datapod* represents an optimized cluster of consumers (*dataclients*) and a producer (*dataserver*) of a replica whose membership varies as a function of the *utility* that the replica brings to the cluster.

The CM maintains network monitoring information about the cache network. Technically, this information is collected by distributed network monitoring tools and reported to a central server, e.g., the CM, as introduced in Section 3.3. When a CA, C , requests a data object d remotely, the CM looks up the catalog for *datapods* that host a replica of d ; if such *datapods* exist, the CM respond to C with the address of the

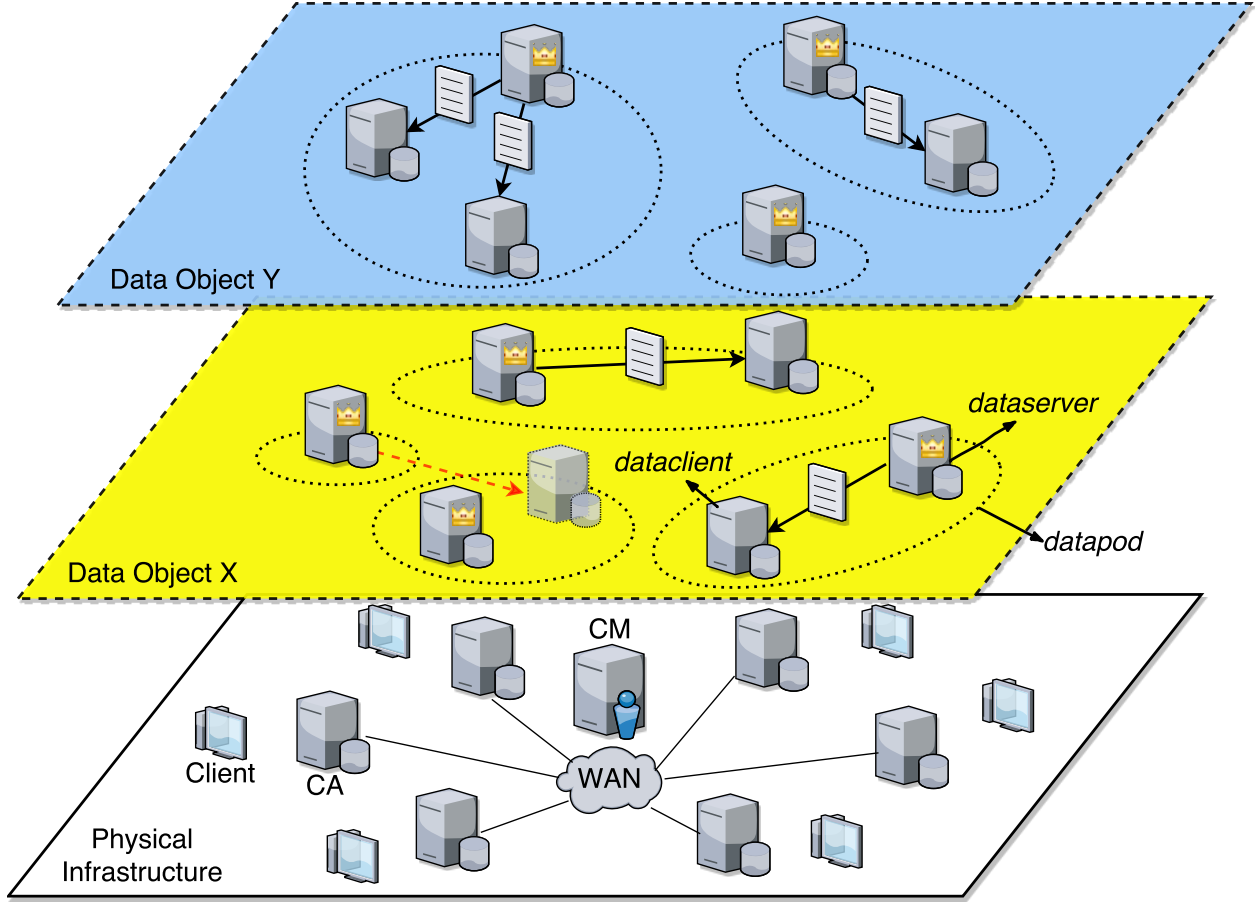


Figure 4.3: CACHALOT architecture. The bottom layer is the physical infrastructure in which CAs are interconnected via WAN and co-located with clients. The CM coordinates cache operations among CAs. The upper layers are *datapod* the topology of different data objects. CAs with a crown are the *dataservers* and *dataclients* fetch replicas from the *dataserver* of *datapods*. Note that a *dataclient* of one *datapod* may migrate to another *datapod* (shown in the middle layer) due to changes in replica distributions and network conditions.

datapod P that incurs the least time to transfer the replica to C together with the aggregate retrieval history of d . Logically, C joins the P after fetching the replica from the *dataserver* of P , denoted by S .

When C joins P , S updates its membership and estimates the available throughput to the new *dataclient* based on the network monitoring information provided by the CM. The *dataserver* also maintains retrieval history of its replica, which includes retrieval frequency and recency (time elapsed since last retrieval) of the replica for each *dataclient*. S uses the retrieval history together with the metadata of data objects, e.g., size, number of replicas, as the metrics to run the cache algorithm and make decisions on the retention of replica in cache. Upon the eviction of a replica, the *datapod* P is dismissed, and thus S requests the CM to delete its record. It is also possible that the cache algorithm triggers the migration of *dataclients* among *datapods* in

observation of changes in network conditions or replica distribution. We will describe this process in detail later in this section.

4.2.3 Network-Aware Cache Algorithm

Following, we describe the network-aware caching algorithms adopted in CACHALOT as explained in Algorithm 1–4 in the form of pseudo code.

Algorithm 1: `insert` function that determines whether to insert a data object on CA

```
/* Note: self is the cache node that invokes the cache algorithm.
   Some implementation details are omitted due to space
   constraint. */
```

Func `insert` (*key*, *obj*) :

```
1  pod ← cm.getPod (key)
2  if pod ≠ ∅ and ¬filter (obj) then
   | return
3  self.evictForSpace (obj.size)
4  pod ← cm.createPod (obj)
5  pod.replica ← obj
6  pod.util ← calcUtil (pod)
7  self.pods ← self.pods + {pod}
```

The network-aware cache algorithm seeks to retain in cache data objects that provide high utility to the entire system. Since there is a one-to-one mapping between a replica and a logical *datapod*, the formation and dismissal of a *datapod* are fulfilled by the insertion and eviction of a replica, respectively. In addition, as explained earlier in this section, the topology and membership of a *datapod* are determined by the network condition observed by its *dataclients*. Hence, we first introduce an utility function that captures the utility of a data object as a function of its generation cost (i.e., computational cost for generating a replica of the data object in units of time), transfer cost (i.e., cost for transferring a replica from a *dataserver* to a *dataclient* in a *datapod*), and usage (e.g., retrieval frequency and recency). This utility function, as formally presented below, drives all the caching and replication decisions on each CA.

$$U(d) = \sum_{m \in M} \frac{f_m^d \cdot \min\{g^d, t_{\kappa_m}^d\}}{s^d \cdot r_m^d \cdot t_m^d} \quad (4.5)$$

The *utility function* is built upon a *cost/benefit* model (Cao and Irani, 1997) and extended to consider dynamic network conditions in the cache network. It is invoked by the *dataserver* of each *datapod* to evaluate the utility of data objects in cache. From the standpoint of a *dataserver*, we use d and M to represent the data object being evaluated and the set of *dataclients* belonging to the *datapod* centering around the *dataserver*. The utility of a data object is the sum of its utility to each *dataclient* in the *datapod*, i.e., $m \in M$. Overall, the numerator and denominator of the function represent the *benefit* and *cost* for caching the data object d in the current *dataserver*, respectively. The *benefit* is a product of two factors. f_m^d represents the frequency of d being accessed by m , implying the popularity of d . The second term indicates the *opportunity cost* for not caching d in the current *dataserver* – if d not cached in the *dataserver*, the *dataclients* are forced to either execute the associated computational job to generate it, which incurs g^d time units to complete the job, or fetch a replica of d from the nearest *datapod* κ_m , which incurs $t_{\kappa_m}^d$ time units to transfer the replica over the network; each *dataclient* will opt for whichever incurs the less time cost for acquiring a replica of d . The *cost* is a composite of size, access recency, and time cost for transferring the replica from the *dataserver* to each *dataclient*, which are denoted by s^d , r_m^d , and t_m^d , respectively. The access recency r_m^d is formatted as the time elapsed (in seconds) from the last access to d by m . As suggested by the utility function, a replica of d is preferable in cache if it is small in size and recently accessed; in addition, it is worthwhile to cache a replica of d in the *dataserver* only if the cumulative time cost for serving the replica to the *dataclients* can be balanced out by the *benefit*.

The utility function is at the heart of all caching activities. Upon receipt of a replica, a CA decides whether to insert the replica into the local cache and form a new *datapod* as illustrated in Algorithm 1. To spare space for the new replica, the CA may need to evict other replicas from the local cache and thus dismiss the associated *datapod*. To make the eviction decision, it first calls the `filter` function (Algorithm 3) to evaluate if the new replica can potentially contribute extra utility. Specifically, the CA forms a hypothetical *datapod* for the new replica (Line 2 in Algorithm 3) and invokes the `calcUtil` function (Algorithm 4) to calculate its potential utility using the aforementioned utility function. If the potential utility surpasses the utility loss caused by evicting replicas with the least utility (Line 1–5 in Algorithm 3), CA moves forward to evict the least-utility replicas (Line 3 in Algorithm 1) and then insert the new replica (Line 4–7 in Algorithm 1).

To retrieve a data object, the CA first looks up the local cache by the key of the data object (Line 1 in Algorithm 2) – if a *datapod* is located, i.e., a replica of the target data object exists, the CA re-calculates the

utility of the replica (Line 3 in Algorithm 2) and returns the replica immediately. If the target data object is not found in the local cache, the CA opts to fetch it from the remote CA with a replica of it in cache. Specifically, it queries against the CM for the nearest *dataserver* of a *datapod* holding a replica of the target data object (Line 5 in Algorithm 2). Note that we define the distance between CAs as the time spent for transferring a single unit of data. If such *datapod* does not exist, the CA returns an empty response as the hint for the caller (i.e., the local client) to executing the associated job in order to generate the desired data object. Notably, the algorithm is observant of the trade-off between re-executing the job and reusing a remote replica (Line 7-10; when fetching a remote replica incurs higher cost, the algorithm raises a dummy *cache miss* to force the re-execution of the associated job rather than unnecessarily stress the network by reusing the remote replica. This step avoids the unnecessary caching of data objects that generates massive output data instantly, thus achieving efficient utilization of cache space and network bandwidth. Lastly, the `insert` function is invoked for every remote retrieval as an attempt to replicate the data object locally (Line 11 in Algorithm 2).

Algorithm 2: `retrieve` function for retrieving a data object by key on CA

```

Func retrieve (key) :
1   if key  $\in$  self.pods then
2       pod  $\leftarrow$  self.pods.getPod (key)
3       pod.util  $\leftarrow$  calcUtil (pod)
4       return pod.replica
5   pod  $\leftarrow$  cm.getPod (key)
6   if pod =  $\emptyset$  then
7       return  $\emptyset$ 
8   t  $\leftarrow$  pod.server.estimateDelay (key)
9   g  $\leftarrow$  cm.getExecTime (key)
10  if t > g then
11      return  $\emptyset$ 
12  obj  $\leftarrow$  pod.server.retrieve (key)
    insert (key, obj)
    return obj

```

The membership of *datapods* dynamically changes in response to three events: 1) upon retrieval of a replica, a *dataclient* may replicate the received data object, forming a new *datapod* on its own and leaving

the one it belonged to; 2) upon eviction of a replica, the associated *datapod* is dismissed and thus *dataclients* of this *datapod* are forced to join others; 3) more commonly, a *dataclient* observes changes in network conditions and migrates from one *datapod* to another accordingly for shortened distance to the *dataserver* that holds the desired replica. In the last case, a *dataclient* can learn from the CM of the nearest *datapod* by calling the `getPod` function in a passive manner during the retrieval of a remote replica (Line 5 in Algorithm 2). In the implementation, the CM maintains and periodically updates the pairwise distance of CAs in a number of binary heaps indexed by the address of each CA, in which the nearest CA is always at the root

of the corresponding binary heap. To find the nearest *datapod*, a *dataclient* locates the binary heap by its own address and keeps checking the root of the binary heap until a CA that hosts the desired data object is found.

Algorithm 3: *filter* function for filtering out incoming data objects with low utility on CA

Func *filter* (*obj*) :

```

1  pod  $\leftarrow$  cm.createHypoPod (obj)
2  gain  $\leftarrow$  calcUtil (pod)
3  if gain = 0 then
    | return false
4  loss  $\leftarrow$  estimateEvictionLoss (obj.size)
5  return gain > loss

```

Algorithm 4: *calcUtil* function for calculating the utility value of data object on CA

Func *calcUtil* (*pod*) :

```

1  util  $\leftarrow$  0
2  obj  $\leftarrow$  pod.replica
3  for c  $\in$  pod.clients do
4  | g  $\leftarrow$  cm.getExecTime (obj.key)
5  | sp  $\leftarrow$  cm.getSecondaryPod (obj.key)
6  | t1  $\leftarrow$  pod.server.estimateDelay (obj.key)
7  | t2  $\leftarrow$  sp.server.estimateDelay (obj.key)
8  | benefit  $\leftarrow$  c.freq * min (g, t2)
9  | cost  $\leftarrow$  c.recency * obj.size * t1
10 | util  $\leftarrow$  util + benefit / cost
11 return utilm

```

4.2.4 Computation Sharing

Data-intensive jobs are typically time- and resource-consuming due to the potentially massive I/O and computational work. These jobs are the norm in the scenarios considered in CACHALOT. Particularly, concurrent requests for the output data of such jobs tend to incur a burst of cache misses and duplicate job execution. That is, when the output data of a job not yet available in CACHALOT, concurrent submissions of

the job will cause cache misses and trigger multiple job executions simultaneously, although only a single execution is needed, and other job submissions can share its result in the optimal case. Furthermore, the duplicate job executions at geo-distributed resources may unnecessarily replicate a data object, wasting the valuable cache space. The mechanism we have introduced by far can hardly avoid the duplicate job executions triggered by such concurrent requests.

Inspired by the *delay scheduling* technique proposed in (Zaharia et al., 2010a), we devise a *computation sharing* mechanism in CACHALOT that delays concurrent requests of identical jobs but only allow one request to hit the system – if it results a cache hit, the system lets in the rest of concurrent requests to reuse the cached output data; otherwise, CACHALOT responds with a cache miss to trigger the job execution and keeps the other requests awaiting the output data to be generated. In specific, the CM takes explicit hints from clients to mark data objects to be generated by jobs in execution; for each request for such data objects, a CA registers a *callback* function on the CM, which fetches the requested data objects when they are available in cache; once the associated job finishes, the client submits the output data to CACHALOT for caching and the CM invokes the associated *callback* functions to deliver the data object to its awaiting clients. As demonstrated later, this mechanism has a significant impact on the performance of CACHALOT.

4.3 Evaluation

We evaluate the performance of CACHALOT through simulation using both synthetic and real-world data sets. We consider two user-driven performance metrics: *cache hit rate* and *job completion time saving*. The *cache hit rate* is the percentage of data accesses satisfied by cache; the *job completion time saving* for each job is mathematically defined as follows.

$$S_j = 1 - \frac{c_j \cdot I_j}{e_j} \quad (4.6)$$

We use c , e and I to denote cache access time, job completion time for job j and whether the job j has a cache hit, respectively. Intuitively, the *job completion time saving* is the fraction of job completion time saved as a result of a cache hit as compared to fully executing the job. We have developed an event-based simulation³ using SimPy⁴.

³Cachalot simulator: <https://github.com/dcvan24/cachalot>

⁴Simpy: <https://simpy.readthedocs.io/en/latest/index.html>

4.3.1 Experiment Setup

4.3.1.1 System Configuration and Environment

In the synthetic data set, we simulate a cache network with 100 CAs. The total cache capacity available in the network is 20% of the total size of unique data objects in the workload. The bandwidth of network links follows log-normal distributions (Crow, 1987) with mean $\mu \in \{1, 2, 3, 4, 5\}$ and standard deviation $\sigma=1$, ranging between 50 and 600Mbps.

The real-world data set consists of network bandwidth traces obtained from ExoGENI, which has more than 14 sites distributed worldwide and connected via more than 10 network providers. To simulate a representative WAN environment, we collected the bandwidth statistics using `iperf3` during the week of July 22–29, 2017.

In both data sets, we assume network latency and packet loss are negligible and computing resources are infinite. All experiments start with a cold cache on each CA in the network.

4.3.1.2 Workloads

To drive our evaluation, we use a synthetic and a real-workload data set.

Our synthesized workloads consist of 10^6 computational jobs with 10^5 unique jobs. Both job popularity and input data size distribution follow a Zipf distribution with $\alpha \in \{0.1, 0.3, 0.5, 0.8, 1.2\}$ which is consistent with characteristics of data-intensive applications in production environments (Breslau et al., 1999; Fricker et al., 2012; Reiss et al., 2012; Chen et al., 2012a). Input data set sizes range between 1.25MB and 125GB which is large enough to cover a broad range of requirements. We assume that the execution time and output data size of jobs are proportional to their input data size which is representative of the vast majority of use cases driving our work. Finally, inter-arrival time of job submissions follows a Poisson distribution with $\lambda=200$ seconds.

To gain insight into the performance of CACHALOT against real workloads, we use the OpenCloud⁵ data set. This data set consists of a 31-month log of Hadoop jobs running on a research cluster in the Carnegie Mellon University and contains 19,198 unique jobs. We have created a workload with 200,000 jobs following the job distribution in this data set.

⁵OpenCloud trace: <http://ftp.pdl.cmu.edu/pub/datasets/hla/dataset.html>

4.3.1.3 Baseline Algorithms

We compare the network-aware algorithm in CACHALOT against three cache algorithms adopted in state-of-the-art distributed computing platforms as below:

- **Least-Frequently-Used (LFU)** favors frequently accessed data objects in cache
- **GreedyDual-Size (GD)** (Cao and Irani, 1997) extends LFU by favoring small-sized data objects that are frequently accessed
- **Nectar** (Gunda et al., 2010) adopts a *cost/benefit* model and favors data objects with high *benefit* but low *cost*. The *benefit* is defined as a product of access frequency and completion time savings, while the *cost* is a product of data size and time elapsed since the data object was last accessed (recency). Notice that Nectar does not take into account network factors

Since both LFU and GD are frequency-based algorithms and GD outperforms LFU, we only show results of GD. We also consider two common replication strategies:

- **Single-copy**: there can be only a *single copy* for each data object existing in the system at any point of time.
- **Replication-based**: data objects can be replicated across CAs with or without constraints.

4.3.2 Synthetic Data Set

We start our evaluation with the synthetic data set.

4.3.2.1 Network-Awareness and Performance Breakdown

We investigate the impact that each mechanism introduced in CACHALOT has on the two performance metrics. We also consider the performance of CACHALOT under three different replication-based strategies: relaxed, filtered and random. In the *relaxed replication* strategy, a CA replicates every data object it retrieves. This strategy is common in production environments. *Filtered replication* relies on the *filter function* introduced in Section 4.2 to make replication decisions. In the *random replication*, a CA makes replication decisions following a uniform random function. Figure 4.4 shows the results of this experiment.

It is observed that when CACHALOT does not take into consideration network monitoring information (static network in Figure) the completion time saving obtained is less than 20% which is significantly lower as compared to all the other cases. Notably, incorporating network information into the scenario with the single-copy replication strategy results in an improvement of almost 100% and 50% in completion time saving and hit rate, respectively. Intuitively, without network information CACHALOT is unable to adjust its replication and caching strategies to network congestion conditions resulting from transferring data across CAs. Our experiments show that in the absence of network information CACHALOT exacerbates network congestion by sending data over congested network links, thus negatively impacting the completion time of jobs.

We then investigate the performance of CACHALOT under the filtered, relaxed and random replication-based strategies. These strategies present interesting trade-offs between data availability, i.e., larger number of replicas, and cache capacity efficiency. Both the relaxed and random replication strategies exhibit comparable performance for both metrics with hit rate under the random strategy being slightly better (25%). This can be explained by the reduction in effective cache capacity resulting from aggressive replication. The filtered replication strategy yields additional 6% time saving in average as compared to the single-copy strategy. Recall from Section 4.2 that the *filter function* in CACHALOT only replicates data when there is a significant gain in *utility*, hence it only trades off an acceptable amount of cache capacity for improved data locality and network performance. We conclude that the *filter function* introduced in CACHALOT offers the best performance overall as compared to the other strategies.

Finally, we evaluate the impact that the optimization mechanism *computation sharing* has over both user-driven performance metrics. Recall that this optimization effectively delays concurrent requests for data objects soon to be available in the cache network. This technique yields roughly 65% and 18% improvement on job completion time saving and hit rate, respectively. Note that without this optimization, CACHALOT would blindly replicate the same data objects, thus hindering the efficient use of network resources in the system.

4.3.2.2 Impact of Network Bandwidth

In the following experiment, we investigate the effect that network bandwidth has over CACHALOT as compared to the baseline techniques. Figure 4.5 depicts the results of this investigation. Note that [s] and [r] refer to single-copy and relaxed replication strategies, respectively.

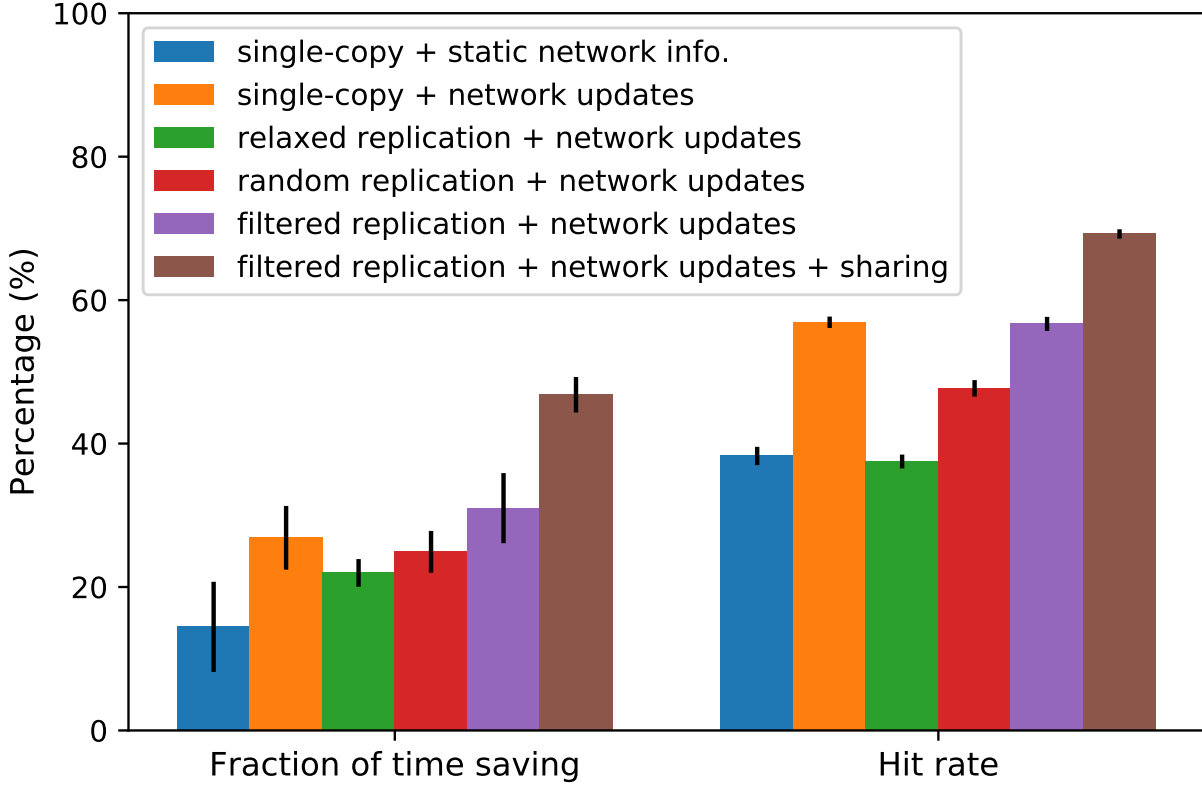


Figure 4.4: CACHALOT cache algorithm performance breakdown

Notably, the time savings achieved under the single-copy replication strategy increases as available bandwidth increases. This is because single-copy algorithms take advantage of increased network bandwidth to transfer remote replicas. In contrast, replication-based algorithms achieve higher local cache hits. Nevertheless, they experience worse time savings due to the reduced effective cache capacity.

As observed, all the other strategies are more or less insensitive to variations in network bandwidth except under extremely constrained conditions (50 Mbps). CACHALOT achieves up to 50% completion time saving as compared to up to 30% and 25% for Nectar[s] and GD[s], respectively, thus demonstrating its ability to efficiently leverage network information.

Similarly, the hit rate for CACHALOT increases with increasing available bandwidth. This is the result of CACHALOT adapting its replication strategy to bandwidth variations while the other strategies are network-agnostic. More importantly, CACHALOT achieves up to 75% hit rate followed by GD[s], GD[r], Nectar[s] and Nectar[r] with 62%, 51%, 50% and 40%, respectively. In-depth analysis of the data shows that CACHALOT replicates more aggressively under constrained conditions and more conservatively otherwise.

When bandwidth is constrained, it trades cache capacity for data locality by keeping popular data objects locally and avoiding data transfers.

Moreover, CACHALOT saves up to 60% network traffic which is in contrast to 35% and 39% savings achieved by GD and Nectar, respectively. This experiment is included in Figure 4.5. In addition, it is observed that the network traffic savings are indirectly proportional to the available bandwidth. This is by virtue of CACHALOT being network-aware and keeping replicas of large data objects local to clients, therefore reducing traffic into the network.

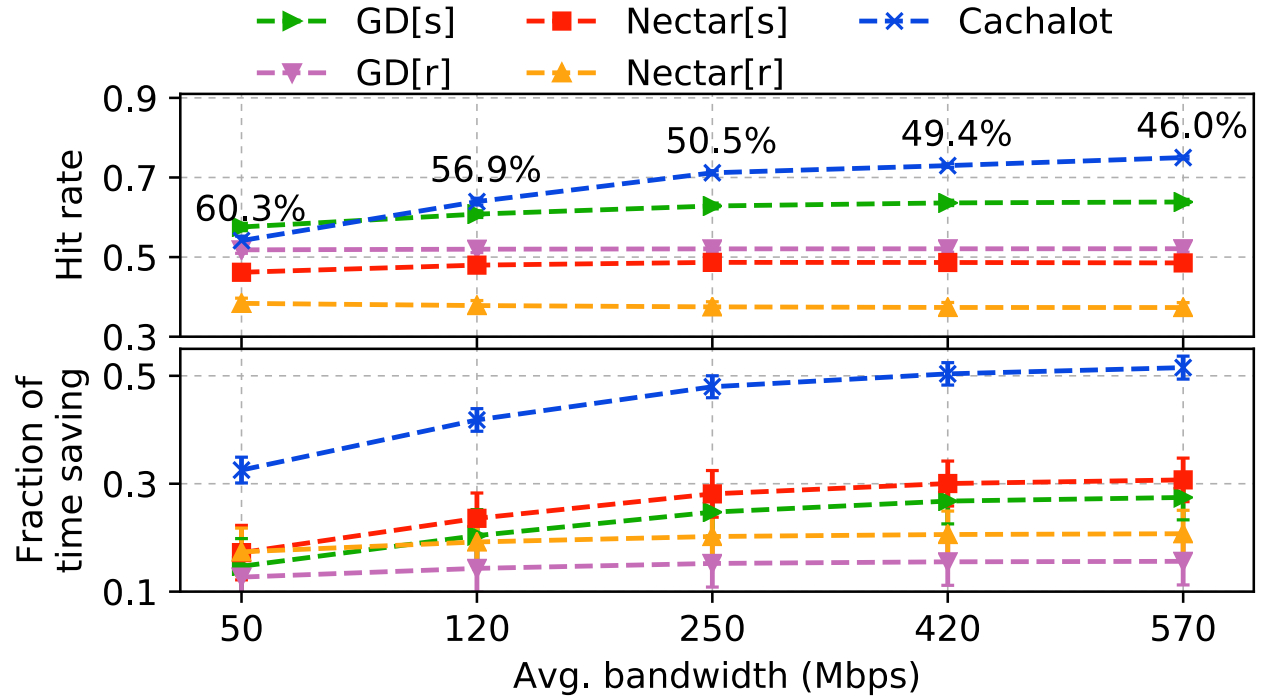


Figure 4.5: Performance with varying bandwidth

4.3.2.3 Stability of *datapods* and replicas

To confirm our previous observation and assess the stability of *datapods* (replicas) in Figure 4.6(a) and Figure 4.6(b), we plot the cumulative distributed function (CDF) of the lifetime of *datapods* and their cardinality, for various values of network bandwidth. As observed earlier, under constrained bandwidth conditions, CACHALOT replicates more aggressively resulting in larger number of *datapods* (See Figure 4.6(a)) with shorter lifetime (See Figure 4.6(b)).

Figure 4.6 also includes graphs for different replication-based algorithms. Single-copy algorithms maintain one long lived replica (*datapod* in CACHALOT) for each data object as compared to replication-

based algorithm. Furthermore, this strategy can lead to the creation of network *hot spots* which have detrimental impact on the overall performance of geo-distributed and data-intensive environments. We do not include graphs for single-copy algorithms due to the lack of space.

As shown, GD[r] creates the largest number of replicas; 40% data objects have more than 5 replicas on average. However, most replicas have a very short lifetime. Likewise, although Nectar[r] creates much fewer replicas than GD[r], their average lifetime is also short. In comparison, CACHALOT effectively trades replica availability by intelligently controlling the creation of *datapods* based on the overall utility that they bring into the system. This observation is particularly important for reducing the management overhead in production environments.

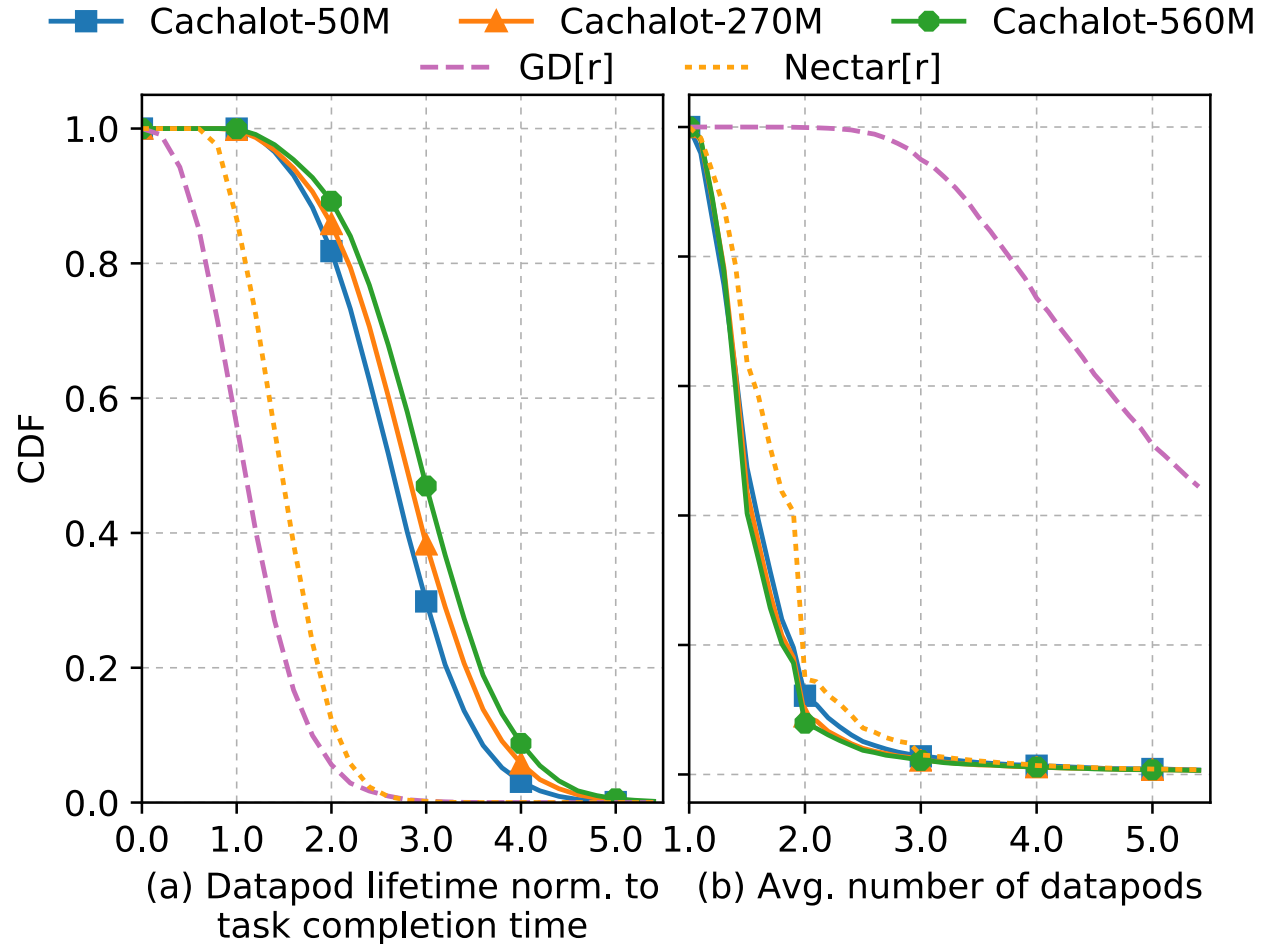


Figure 4.6: Distribution of *datapod* number and lifetime

In the next experiment, we evaluate the impact that the data size distribution has on the performance of CACHALOT. Figure 4.7 depicts hit rate and average time saving as a function of α for GD[s], GD[r],

Nectar[s], Nectar[r] and CACHALOT. The larger α the larger the fraction of small data objects. The graph shows that the hit rate is less sensitive to the data size distribution under CACHALOT as compared to the other techniques. This follows intuition since the data size factor is outweighed by the network factor in the utility function of CACHALOT.

We also observe that the job completion time saving decreases when $\alpha=1.2$ for all algorithms including CACHALOT. This is due to the fact that the majority of data objects are small and hence their corresponding jobs have short execution times. As a result, these jobs observe marginal gain from caching since the time for fetching data from remote CAs may be comparable to the time it takes to generate the data. For instance, GD and Nectar cache aggressively when data objects are small (reflected on the increasing hit rate in the figure), but fail to produce significant time saving. To make matters worse, they generate more network traffic and fail to efficiently use network bandwidth. In contrast, CACHALOT outperforms all baseline algorithms achieving up to 50% completion time saving for all values of α due to its unique ability to take into account network monitoring information and demand.

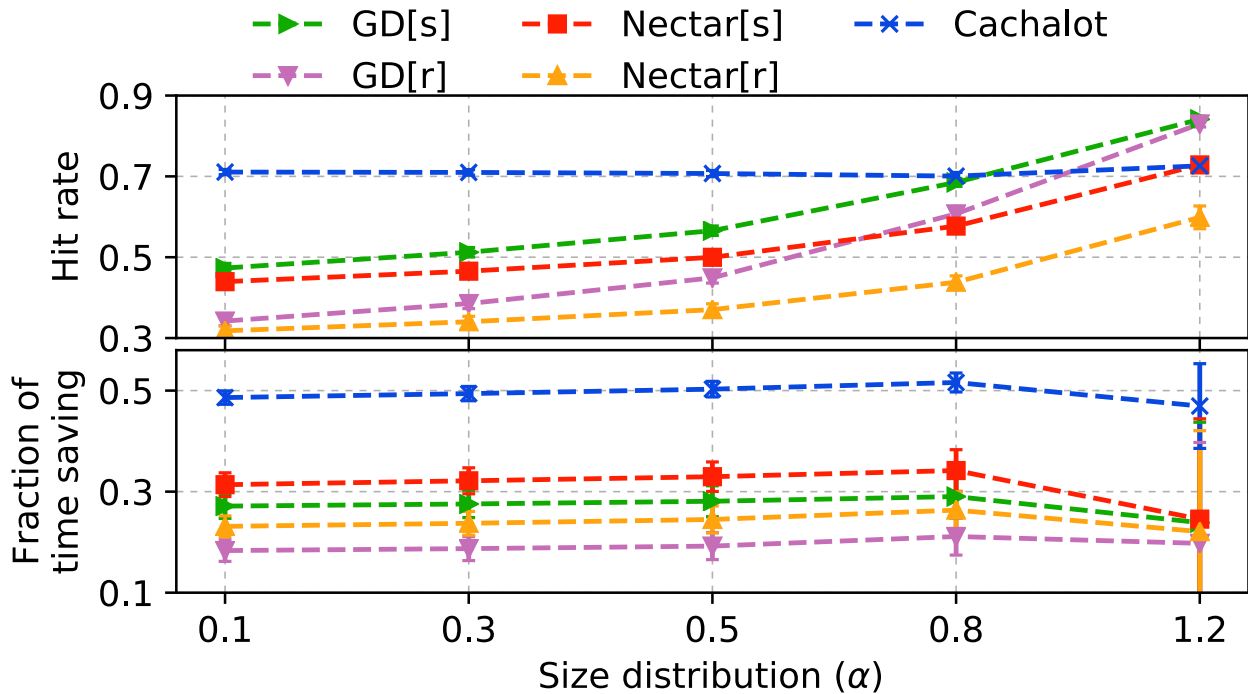


Figure 4.7: Performance with varying sizes

Naturally, as some data objects become very popular, it becomes more difficult to balance the use of cache and network resources in the system. To investigate this observation, we evaluate the performance of the replication algorithms as a function of data popularity. More specifically, we vary α to skew the popularity

of data objects. Figure 4.8 depicts the results of this evaluation. As expected, since caching top-ranked data objects can fulfill most data requests, hit rate increases as increasing function of α . Particularly, algorithms with replication-based strategy outperform algorithms with single-copy replication strategy for large values of α ($\alpha=1.2$). This follows intuition since aggressive replication effectively improves data locality of top-ranked data objects and thus reduces the need for data transfers. This reasoning is confirmed in the plot depicting the *fraction of local hits* in Figure 4.8 which shows that both GD[r] and Nectar[r] have a higher fraction of local cache hits as compared to CACHALOT, Nectar[s] and GD[s]. This result is important because it demonstrates that CACHALOT outperforms the baseline algorithms in hit rate and average job completion time by virtue of its network awareness as demonstrated by the modest fraction of local cache hits (less than 20%). We conclude that CACHALOT is able to more effectively utilize cache and network resources while observing user-driven metrics by deciding when to replicate (via the utility function) and which replica to select in a network-aware fashion.

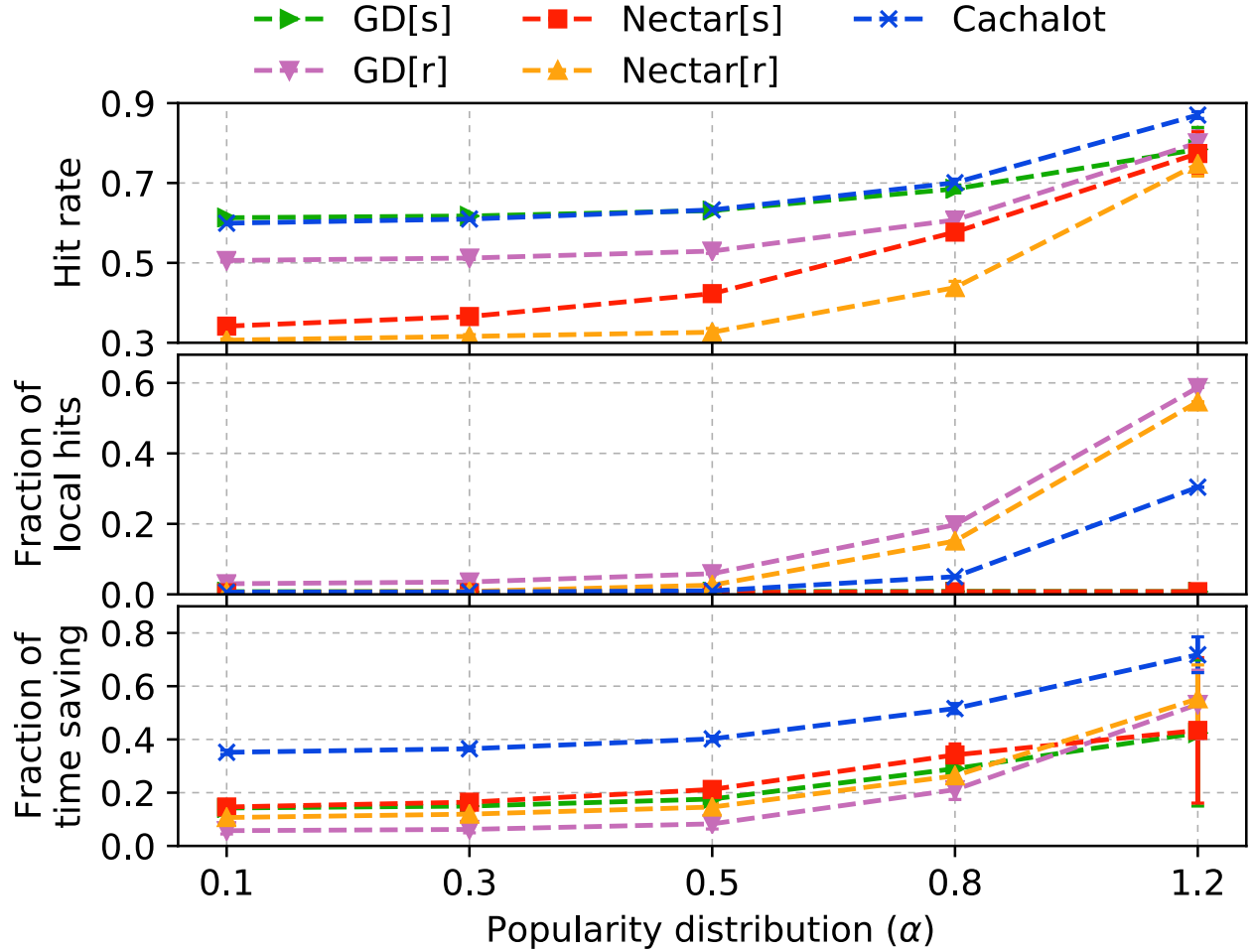


Figure 4.8: Performance with varying popularity distribution

4.3.3 Real-World Data Set

Finally, we simulate a real-world environment by merging the workload characteristics of the OpenCloud data set – to simulate workload – with the network bandwidth collection from ExoGENI – to simulate a realistic network as explained in Section 4.3.1. Figure 4.9 depicts four dimensions of the data set: output data size, popularity distribution of data, network bandwidth and job execution time. We observe that the majority of jobs in the workload have relatively short execution times and generate small data output. The size and popularity of data objects follow Zipf-like distributions. In contrast with the assumption made in our earlier experiments, the job execution time is independent of the output data size. The network consists of 14 CAs interconnected with network links with an average bandwidth of 215Mbps.

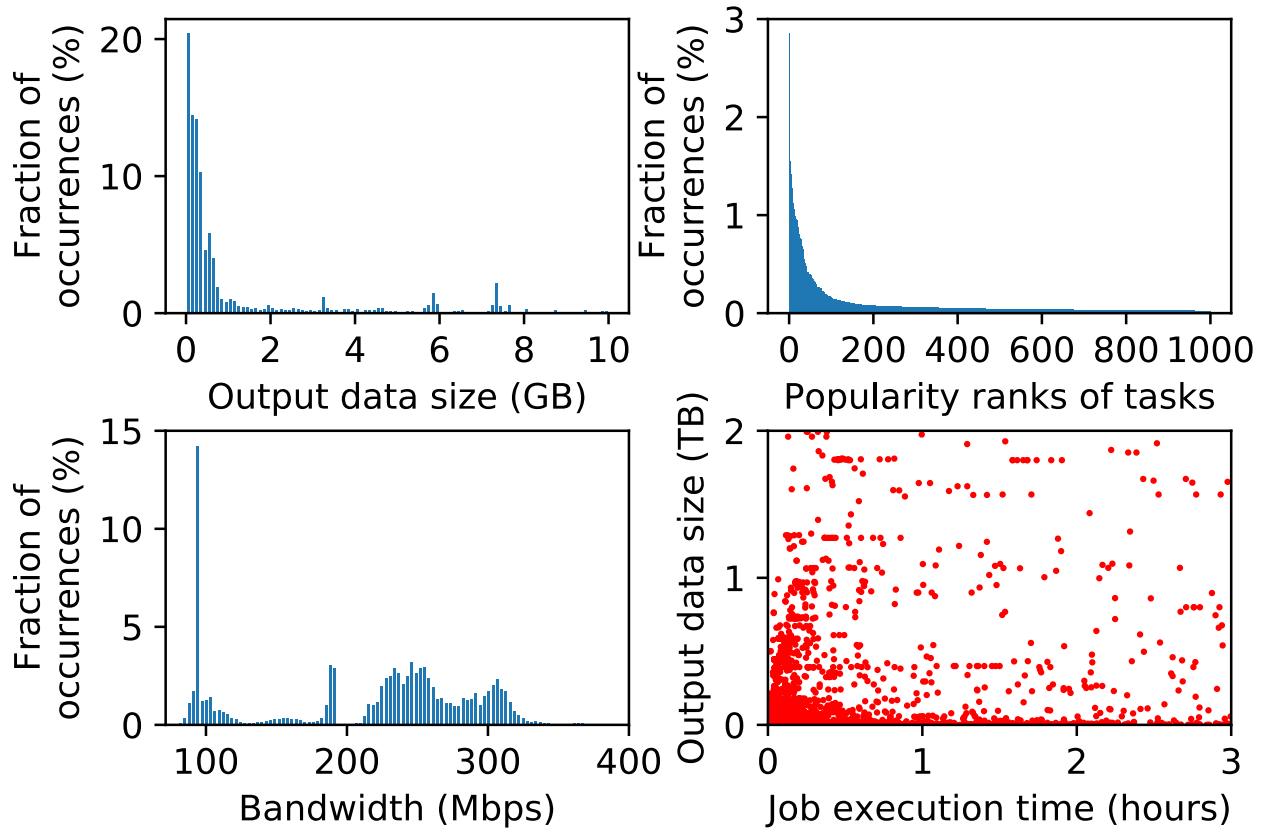


Figure 4.9: Characteristics of OpenCloud and ExoGENI trace data set

4.3.3.1 Impact of Cache Capacity on Performance

In a production federated deployment there is no control over the amount of cache resources that each site (CA) contributes with to CACHALOT. In this experiment we study the impact that total cache capacity

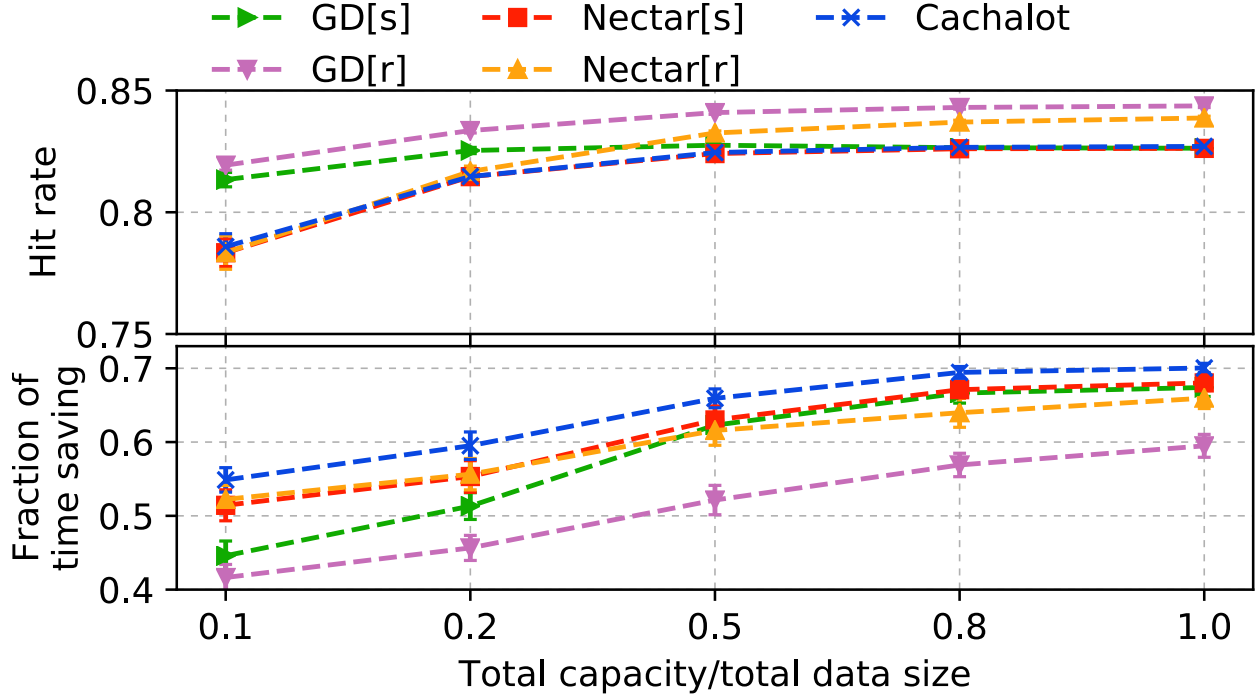


Figure 4.10: Performance with varying capacity

has on performance for all the algorithms. Our results are plotted in Figure 4.10. We make three observations. First, all algorithms perform well in terms of hit rate for a sufficiently provisioned cache (hit rate varies between 80% and 85%). This is due to the fact that most popular data objects are small in size hence the algorithms take advantage of statistical multiplexing of cache resources even when the cache capacity is limited. In particular, replication-based strategy can efficiently utilize storage cache capacity by packing replicas of small data objects. Second, CACHALOT exhibits the lowest hit rate. A deeper look into the results shows that CACHALOT tends to evict jobs that although small in duration generate large output (bottom-right sub-figure in Figure 4.9). Notice that these jobs cannot take advantage of caching due to their prohibitive transfer cost which in turn exacerbates congestion in the network. Third, as a consequence of our previous observation, CACHALOT achieves the largest completion job saving as compared to the other algorithms. GD performs the worst in terms of completion time saving since it favors caching small data objects regardless of their overall completion time.

4.4 Chapter Summary

In this chapter, we have introduced CACHALOT, a geo-distributed cache network for caching repeatedly used output data sets of frequent data-intensive jobs, thus avoiding unnecessary job re-execution. Specifically, we have proposed a system design on top of the WAN-based, geo-distributed infrastructure as discussed in the prior chapter, and designed a network-aware cache algorithm that incorporates the dynamic network factors into the algorithmic decisions. Furthermore, we have introduced a computation sharing mechanism that elegantly handles bursts of concurrent requests for identical data objects. The experimental evaluation shows that CACHALOT can effectively save the time and resource wastage on duplicate job execution by caching and dynamically adapt to varying network conditions for high utilization of cache space.

CHAPTER 5: Enable Cost-Aware Scheduling of Applications in a Multi-Cloud Environment

In this chapter¹, we explore the potential of building a geo-distributed computing system on top of multiple public or private cloud platforms and introduce our cloud-agnostic, cost-efficient framework PIVOT in detail. In prior chapters, we focused on geo-distributed, data-intensive applications running on top of a single computing platform. For instance, we have developed RADII on top of ExoGENI as introduced in Chapter 3. However, binding with a single platform imposes a huge limitation on data-intensive applications as the demand for cloud computing continues to grow. We have observed valuable data sets distributed across cloud platforms. On the other hand, every cloud platform provides unique capabilities and service offerings essential to different data analytic applications, such as Google Cloud Spanner² and AWS Redshift³, which force researchers and developers to switch among cloud platforms in order to execute various applications.

The distribution of critical data sets and services across cloud platforms imposes three major challenges for data-driven analyses and applications. Firstly, the heterogeneity in resources, networking, APIs and runtime among cloud vendors prevents applications from scaling out across platforms to leverage the unique services and execute in proximity to data; in other words, an application is limited to a single cloud platform for execution. Secondly, without the ability to scale across cloud platforms in a seamless fashion, applications are forced to initiate data movements across geographical regions and cloud platforms, which may hinder the application performance due to poor throughput over the WAN. Lastly, since commercial cloud platforms monetize egress network traffic, i.e., outbound traffic from a cloud region or platform, cross-region and cross-cloud execution of data-intensive applications can incur prohibitive cloud expenses for massive data transfers.

To address the aforementioned challenges, we have developed PIVOT, a cloud-agnostic framework that creates an abstraction layer over computing, storage, and networking resources distributed across cloud

¹Content of this chapter previous appeared in preliminary form in the following paper:

Jiang, F., Ferriter, K., and Castillo, C. (2020). A Cloud-Agnostic Framework to Enable Cost-Aware Scheduling of Applications in a Multi-Cloud Environment. In NOMS 2020-2020 IEEE/IFIP Network Operations and Management Symposium. IEEE.

²Google Cloud Spanner: <https://cloud.google.com/spanner>

³AWS Redshift: <https://aws.amazon.com/redshift>

platforms to create an illusion of a single monolithic computer, thus hiding the heterogeneity and complexity in many aspects among cloud vendors. PIVOT presents these resources through a unified API with which data processing applications, e.g., genomic workflows, can execute and scale across resources independently of the geo-location and cloud platform where the data is stored and critical services are provided. To achieve this, we have built a middleware mechanism that orchestrates the joint utilization of multi-cloud, geo-distributed resources and services with acceptable performance, while also taking into account cloud expenses it incurs. Specifically, we have developed a simple yet powerful cost-aware resource scheduling algorithm that factors in egress network traffic cost to cope with scenarios wherein applications consume data and services from multiple cloud regions and platforms.

For the rest of this chapter, we first elaborate on the system design and implementation of PIVOT (Section 5.1). Then, we explain the cost-aware scheduling algorithm in detail (Section 5.2). Lastly, we present a thorough evaluation of the proposed algorithm (Section 5.3) and summarize this chapter (Section 5.4).

5.1 System Design

In PIVOT, we seek to support cross-cloud, cross-region execution of data-intensive applications while hiding the complexity of the underlying heterogeneous systems and respecting cost and performance requirements of the application. We achieve this through a cloud-agnostic framework introduced in PIVOT that abstracts virtual infrastructure provided by IaaS across clouds, and a customizable two-level scheduling framework that minimizes cost by optimizing job placement for data locality. In this section, we describe the system design of PIVOT in detail.

5.1.1 PIVOT Application

We first define the application running in PIVOT. Different from applications in most systems, an *application* in PIVOT is composed of a mixture of logically organized *services* and *jobs* instead of a single kind – a *service* is a long-running process that handles ad-hoc requests, e.g., online analytical processing (OLAP) queries; a *job* is a procedure with limited repetitions for processing the assigned workload, e.g., workflow jobs. The *services* and *jobs* of an *application* have dependencies among each other, which are internally represented in a directed acyclic graph (DAG) in PIVOT. Each *service* or *job* can be split into a number of atomic *tasks* in parallel, each of which takes a portion, if not the entirety, of the workload

designated to the *service* or *job*. A *task* specifies the program to execute as well as explicit resource demand for CPUs, RAM, disk space, and GPUs. As discussed shortly in Section 5.1.2.3, the global scheduler of PIVOT performs application scheduling at the granularity of *tasks*.

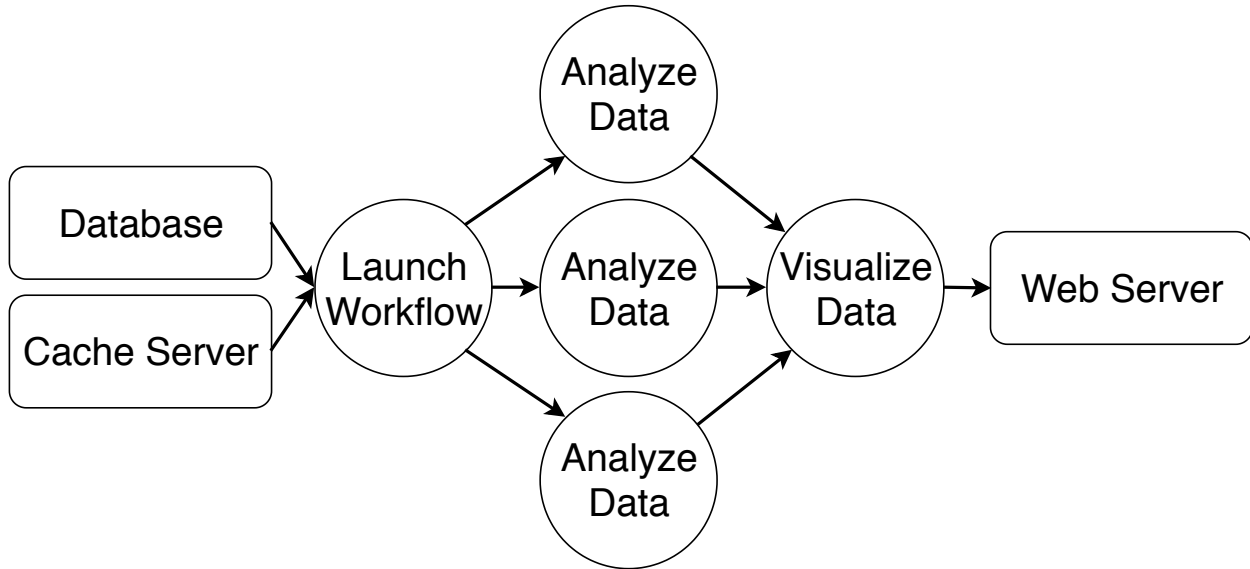


Figure 5.1: An example of PIVOT application

Figure 5.1 shows an example of PIVOT application with both *services* and *jobs*, which supports a repeatable data analytical workflow. The `Database` and `Cache Server` are *services* that feed and cache input data for the workflow, respectively. The `Launch Workflow` is a *job* that starts the workflow, which will be triggered following the precedent *services* being active and stable. The workflow is simplistic in structure, containing a number of `Analyze Data` *jobs* in parallel for data analysis followed by a `Visualize Data` *job* for visualizing the output data of the data analysis. At the last step, a `Web Server` *service* is spun up to render the visual generated by the last *job*.

5.1.2 PIVOT Architecture

The PIVOT architecture is designed with cloud agnosticism at its core as illustrated in Figure 5.2. At the bottom, we have built a *cross-cloud virtual infrastructure* across multiple cloud platforms, in which we have enabled seamless network connectivity across the geo-distributed cloud regions and platforms. On top of that, we have introduced an *abstraction layer*, which abstracts the heterogeneous resource offerings from the various cloud vendors into fine-grained, on-demand resources that can be flexibly utilized by applications. On the basis of the *abstraction layer*, we have developed the *scheduling layer* that employs a pluggable

two-level scheduling framework to schedule computational tasks in consideration of both the system-wide and application-level scheduling goals. At the top, PIVOT exposes a REST API to interact with users and external systems.

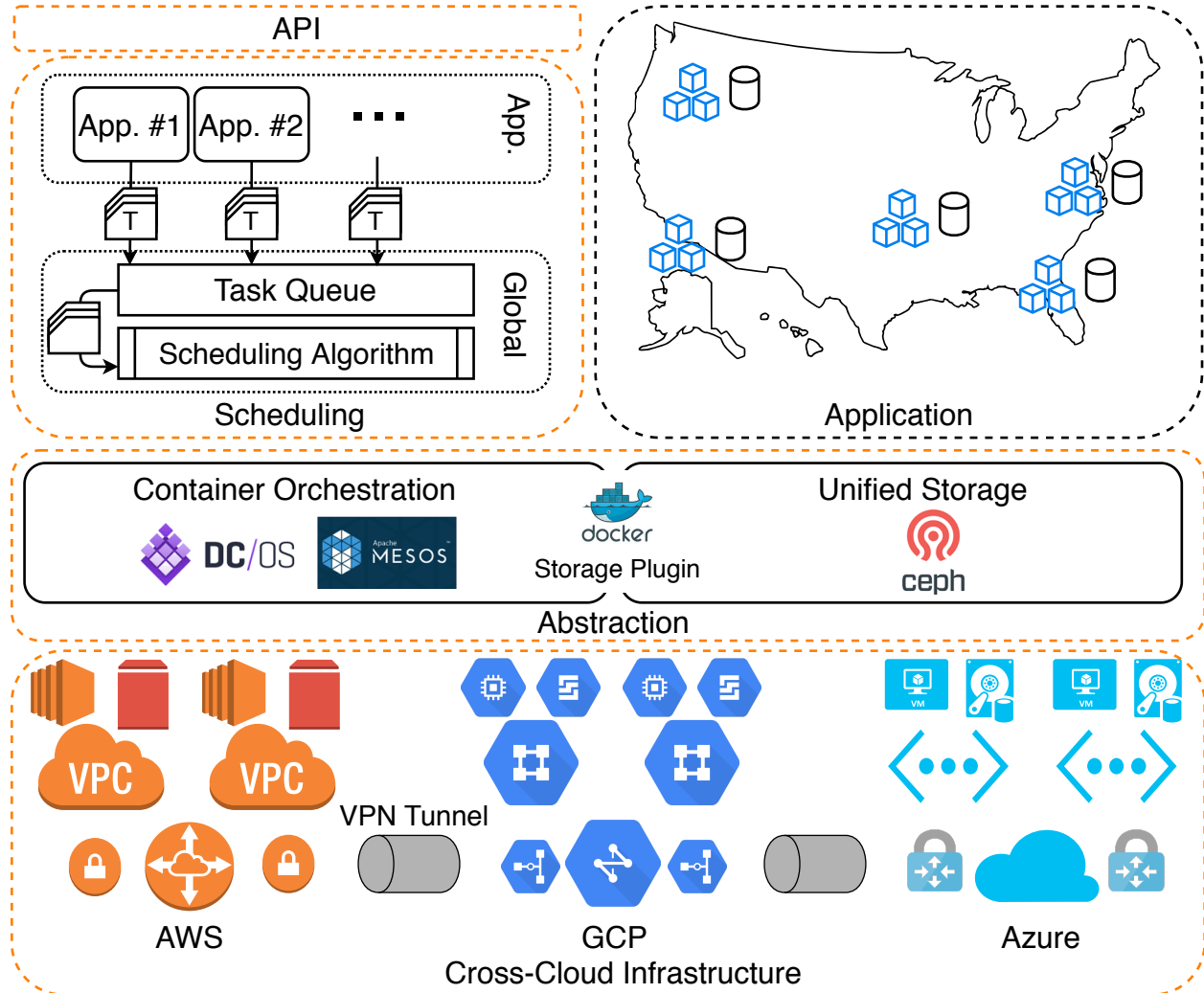


Figure 5.2: PIVOT architecture

5.1.2.1 Cross-Cloud Virtual Infrastructure

The *cross-cloud virtual infrastructure* unifies IaaS resources provisioned in different cloud regions and platforms and offers them to applications in standard units at a fine granularity (e.g., a tenth of a CPU cycle, kilobyte-level memory allocation), creating a federation of resource pools and orchestrating all aspects of resource management and communication across cloud regions and platforms. The virtual infrastructure provisions computing and storage resources to the upper layer, i.e., the *abstract layer*. The computing

resources are backed by VM instances mainly configured with computing units such as CPUs, GPUs and RAM, while the storage resources are VM instances with high-capacity storage devices attached. By default, these geo-distributed VM instances reside in isolated private networks and communicate with each other via the Internet. This default network configuration can lead to suboptimal network performance due to the bandwidth limitation and instability of the Internet. More importantly, it potentially incurs unnecessary expenses for network traffic as most cloud vendors charge for egress network traffic through the Internet at a high rate, which can be avoided for network traffic within a cloud region by using the private network instead. Furthermore, it is hazardous to expose the VM instances to the Internet, which potentially exhibits vulnerabilities to malicious attacks.

To address these issues, we have constructed a virtual *supernet* spanning across cloud regions and platforms to connect the VM instances. To connect VM instances on the same platform, we rely on the virtual private cloud (VPC) or product alike, which is provided by mainstream cloud vendors. Specifically, we have set up a VPC for each availability zone (AZ) in a cloud region and peer up the VPCs in different regions to establish a platform WAN. The major advantage of the platform WAN is that it leverages the specialized network infrastructure provided by the platform and tends to yield performance improvement over the Internet. Moreover, network traffic in the WAN will be charged at a significantly lower rate (up to 12x) or even free, since it is no longer carried over through the Internet. Furthermore, by using VPCs, we have many options to secure VM instances, e.g., restricting their public accessibility, configuring a firewall in front of each VPC. Last but not least, since VPCs are highly customizable, the use of VPCs lends a plenty of opportunities to PIVOT for network customization and optimization, e.g., building an SDN virtual network as we have discussed in Chapter 3.

To federate the platform WAN, we have set up a virtual private network (VPN) tunnel over the Internet between each pair of regional VPCs across platforms, creating an illusion of a single flat *supernet* to applications running on top. Although transferring data over the Internet, the VPN tunnels encrypt the data using AES-256 (Daemen and Rijmen, 2001) and thus secure cross-cloud communications. Moreover, the VPN tunnels can be deployed automatically and dynamically, requiring minimum time and monetary investment and therefore suitable for short-term, dynamic research projects. However, the VPN tunneling can hardly resolve the performance issue with the Internet but may introduce extra delay for data encryption. Alternatively, for a long-term deployment of PIVOT, we can opt to reserve dedicated network circuits between

cloud platforms using cloud services, such as AWS Direct Connect⁴ and GCP Dedicated Interconnect⁵, which requires non-trivial manual configurations and cloud expenses but greatly improves network performance for cross-cloud data transfers.

Built on top of essential IaaS components widely available across platforms, the *cross-cloud virtual infrastructure* is not only limited to the mainstream public cloud platforms but can also incorporate computing clusters offering IaaS services. For instance, the virtual infrastructure can be extended to any OpenStack⁶ deployment on which all the capabilities it requires can be enabled in the form of VM instances. We have experimented such integration with the Chameleon Cloud⁷, which is powered by OpenStack at the backend. That said, it is possible to federate geo-distributed computing clusters on campus to serve as the *cross-cloud virtual infrastructure* for PIVOT.

5.1.2.2 Abstraction Layer

The *cross-cloud virtual infrastructure* lays the foundation for PIVOT, provisioning geo-distributed, standard virtual resources for data-intensive applications. However, there are still substantial technical and business barriers between cloud regions and platforms that impede the applications from scaling across. Hence, we have introduced the *abstraction layer* that abstracts the resources provisioned from disparate cloud regions and platforms, to break the boundaries among the resources and thus enable applications to seamlessly scale across regions and platforms. By design, the *abstraction layer* is composed of the *container orchestration* and *unified storage* components for abstracting the computing and storage resources, respectively.

To abstract computing resources, we leverage the OS-level virtualization technique to encapsulate application components into generic, lightweight user-space units, referred to as containers, which can scale and migrate across heterogeneous, distributed resources with little friction. Specifically, we enclose every *task* into a dedicated container, which is assigned to specific VM instance for execution. Furthermore, we resort to the container orchestration framework to orchestrate containers for application execution. Instead of scheduling containers using the built-in algorithms of the framework, the *abstraction layer* delegates

⁴AWS Direct Connect: <https://aws.amazon.com/directconnect>

⁵GCP Dedicated Interconnect: <https://cloud.google.com/interconnect/docs/concepts/dedicated-overview>

⁶OpenStack: <https://www.openstack.org>

⁷Chameleon Cloud: <https://www.chameleoncloud.org>

the container scheduling to its upper layer, i.e., the *scheduling* layer, by exposing to it the scheduling “knobs” provided by the framework, e.g., the API for assigning a container to a specific VM instance. In the implementation, we use Docker⁸ and DC/OS⁹ as the container runtime and container orchestration framework, respectively.

To abstract storage resources, we have built the *unified storage* on top of the fault-tolerant distributed storage system to store and share data for applications in the form of *volumes*. A *volume* is a data storage unit that stores data produced and consumed by containers. It implements strong consistency and provides global accessibility to the distributed containers throughout the system. Detached from local storage of underlying computing resources, containers use the *volumes* instead to keep persistent states for high scalability and fault tolerance – they can freely scale in and out without being restricted to specific resources; upon failures, they can restore elsewhere from the states stored in the associated *volumes*. A *volume* can also serve as the data sharing medium among containers; it can be attached to multiple containers and allows the containers to read and write data simultaneously with consistency. In essence, we implement the *volume* as an instance of a POSIX-compliant distributed file system, on which containers can perform the identical I/O operations as on a local file system. Under the hood, we select Ceph (Weil et al., 2006a) as the distributed storage system backing the *unified storage* and CephFS¹⁰ to implement the *volumes*. To bridge the gap between the containers and *volumes*, we have developed Docker storage plugin for CephFS¹¹, which maps CephFS instances to *volumes* attachable to Docker containers. In addition, we have also enabled custom *volume* placement using the CRUSH rules (Weil et al., 2006b) to place *volumes* in proximity to containers and thus improve the data locality. Nevertheless, the configuration of CRUSH rules is rigid in the implementation of Ceph, which hinders the dynamic placement of *volumes* in the *unified storage*. Hence, PIVOT largely relies on proper *task* placement to retain data locality for applications but only enforces specific *volume* placement as explicitly specified in an *application* during its initial deployment.

⁸Docker: <https://www.docker.com>

⁹DC/OS: <https://dcos.io>

¹⁰CephFS: <https://docs.ceph.com/docs/master/cephfs>

¹¹Docker volume plugin for CephFS: <https://github.com/dcvan24/cephfs-docker-volume-plugin>

5.1.2.3 Scheduling Layer

The *scheduler layer* is the brain of PIVOT, geared with the logic for properly scheduling *tasks* onto the underlying resources. It only concentrates on the scheduling logic as the *abstraction layer* underneath guarantees that *tasks* can execute, scale, and migrate freely across cloud regions and platforms. Since PIVOT hosts a variety of applications with diverse scheduling goals, we have designed the *scheduling layer* as a modular and extensible *two-level scheduling framework*, which accepts custom scheduling algorithms to accommodate both the system-wide and application-specific scheduling goals.

The framework consists of a single *global scheduler* and an extensible array of *application schedulers* as shown in Figure 5.2. Each *application* has an instance of *application scheduler* for scheduling its *tasks* throughout its lifetime to meet the application-specific scheduling goal. Specifically, it makes decisions on task placement based on its scheduling goal and resource availability in the system. For instance, the default *application scheduler* in PIVOT implements the rudimentary dependency resolution algorithm to ensure the dependencies among *tasks* are not violated. A custom *application scheduler* can be developed by extending the abstract *application scheduler*, in which critical functionalities and API are pre-defined. Every *application scheduler* is installed as a pluggable module in the framework and can be reused by *applications* by creating a dedicated instance of it for each *application*. The *application scheduler* operates in a phased fashion, dispatching *tasks* to the *global scheduler* periodically in observation of current states of *tasks* and resource availability in the system.

The *global scheduler* is responsible for achieving the system-wide scheduling goal. It implements a *task queue* wherein *tasks* submitted by *application schedulers* are queued up for the final dispatch to the underlying layers for execution. The *global scheduler* polls *tasks* from the queue periodically, validating the task placement decisions made by *application schedulers* against the system-wide scheduling goal and resolving conflicts if any; upon any conflict, the *global scheduler* finalizes the task placement decision based on the system-wide scheduling goal and current resource availability. It is possible that the conflict in scheduling goal is unresolvable – in this case, the *global scheduler* fails the involved *tasks* on purpose to let the *application scheduler* adjust the task placement decision and reschedule the *tasks*.

The two-level design of the scheduling framework enhances the versatility and robustness of the *scheduling layer* by offloading application-specific scheduling logic to the *application scheduler* and keeping the simplicity of the *global scheduler*. New application scheduling algorithms can be added to the *scheduling*

layer without interfering with the core *global scheduler*, therefore lowering the risk of system breakdown caused by scheduler failures.

5.2 Cost-Aware Scheduling Algorithm

Running data-intensive applications across geo-distributed cloud regions and platforms tends to create significant financial burden due to the high cost for egress network traffic and resource subscription. On the other hand, it is inevitable to run applications in the cross-cloud, geo-distributed fashion to gain data locality and thus improve the application efficiency. To strike a balance between these two factors, we propose a cost-aware scheduling algorithm for the global scheduling of applications in PIVOT, which aims at saving cloud expenditures for running cross-cloud, geo-distributed applications while improving data transfer efficiency in a best-effort manner.

5.2.1 Problem Definition

We first formulate the problem mathematically. We consider the model of PIVOT application as described in Section 5.1.1, in which there are a number of *tasks* with dependencies among each other.

A set of *tasks* T are placed and executed on geo-distributed VM instances H distributed across cloud regions and clouds. Each *task* $\tau \in T$ has the resource demand d_τ , which is a 4-dimensional vector that includes the amount of CPUs, RAM, disk space, and GPUs. Likewise, each VM instance $\eta \in H$ is also associated with a vector R_η that represents the resource capacity in the aforementioned dimensions. Here we use K to denote the resource dimensionality. With the IaaS service model, subscribing a VM instance η incurs a VM subscription cost of i_η dollars per hour. We use a bit array $u_{|H|}$ to indicate whether η is in use and assume that idle VM instances will be timely shut off to save the unnecessary subscription cost. In addition, transmitting data between η and η' incurs egress network traffic cost $e_{\eta,\eta'}$ dollars per GB. We use a binary matrix $P_{|T| \times |H|}$ to represent whether the data transfer between tasks τ on η and τ' on η' is possible. Lastly, since the problem is a multi-objective optimization, we introduce α and β as coefficients. The problem is formulated as below.

$$\min \quad \alpha \cdot \sum_{\eta \in H} i_{\eta} \cdot u_{\eta} + \beta \cdot \sum_{\tau \in T} \sum_{\tau' \in T} \sum_{\eta \in H} \sum_{\eta' \in H} (P_{\tau\eta} + P_{\tau'\eta'}) \cdot e_{\eta\eta'} \quad (5.1)$$

$$\text{s.t.} \quad \sum_{\tau \in T} d_{\tau k} \cdot P_{\tau\eta} \leq R_{\eta k} \quad \forall \eta \in H, k \in K \quad (5.2)$$

$$\sum_{\eta \in H} P_{\tau\eta} = 1 \quad \forall \tau \in T \quad (5.3)$$

$$u_{\eta} \in \{0, 1\} \quad \forall \eta \in H, \forall v \in V \quad (5.4)$$

$$P_{\tau\eta} \in \{0, 1\} \quad \forall \tau \in T, \forall \eta \in H \quad (5.5)$$

The objective is to minimize the total expense for VM subscription and egress network traffic. The constraint 5.2 is the capacity constraint that limits the total resource demand of tasks placed on a host to the resource capacity of the host in any dimension. The constraint 5.3 ensures that a task can only be placed on one host at any point of time. The constraints 5.4 and 5.5 indicate that U and P are binary vector and matrix, respectively. We recognize the problem as a variant of the MDVBP (Chekuri and Khanna, 1999; Frenk et al., 1990), which is proven NP-hard. Hence, we propose a cost-aware heuristic instead to tackle the problem.

5.2.2 Algorithm Design

As identified in the prior section, the expenses for running data-intensive applications in the cloud primarily comprise the cost for VM subscription and egress network traffic among cloud regions and platforms. Therefore, resource underutilization and excessive cross-cloud and cross-region data transfers will impose a huge financial burden. Cross-cloud and cross-region data transfers tend to exhibit low efficiency due to the constrained bandwidth and high latency between the cloud regions and platforms, therefore slowing down applications in overall. To lower the cloud expenses while improving application performance, we design the cost-aware scheduling algorithm that consolidates applications into the least number of VM instances to improve resource utilization and eliminate unnecessary egress network traffic.

We recognize that the problem of application consolidation is analogous to the vector bin packing (VBP) problem – *tasks* and VM instances are *items* and *bins* in the VBP problem, respectively; and we aim at placing *items* into the fewest *bins* considering sizes of *items* and capacity of *bins*. The problem is an NP-hard problem in general (Karp, 1972), and greedy approximation heuristics, such as first fit (FF) (Panigrahy et al., 2011)

and best fit (BF) (Beloglazov and Buyya, 2010), and their variants have been developed and proven effective in approximating the optimum. We refer to the family of FF and BF algorithms as VBP algorithms, and refer readers to Dósa (2007) and Panigrahy et al. (2011) for the in-depth analysis of VBP algorithms.

In the classic VBP problem, *items* may have different sizes while *bins* are presumably homogeneous in capacity. Therefore, by design, existing heuristics assume that the ordering of *bins* has no impact on the algorithmic result and assign *bins* to *items* in an arbitrary order. However, in the settings of PIVOT, VM instances are distinguished by a number of factors such as VM subscription cost, egress network traffic cost, inbound and outbound network bandwidth, data locality, among others. Hence, the ordering of VM instances entails cost and performance implications for task assignment – it is preferable to place a *task* onto a VM instance in proximity to its input data, which we refer to as the *anchor*, to gain data locality; if such *anchor* is unavailable for placing the *task*, the algorithm searches for another VM instance radially surrounding the *anchor* based on a function of resource availability, VM subscription cost, egress network traffic cost, and network bandwidth from and to the *anchor*. Intuitively, by scores calculated by the function, the VM with the highest score will be elected for the task placement, which potentially yields optimal cost saving and data transfer efficiency.

Algorithm 5: Cost-aware scheduling

```

Func schedule ( $T, H$ ) :
1   $G \leftarrow \text{GroupTasks}(T)$ 
2  for  $g \in G$  do
3       $g \leftarrow \text{SortTasks}(g)$ 
4       $H \leftarrow \text{SortVMs}(H, \theta)$ 
5      FirstFit( $g, H$ )
6  return  $T$ 

```

Following this intuition, we design our cost-aware scheduling algorithm as shown in Algorithm 5: 1) during each scheduling epoch, *tasks* to be scheduled are divided into groups based on the location of their input data, i.e., *anchor* (Line 1); 2) for each group, the algorithm sorts the *tasks* in the group (Line 3) and VM instances (Line 4) with reference to the *anchor*, respectively; the algorithm applies the FF algorithm to pack *tasks* in each group into VM instances in order (Line 5). Following, we describe each step of the algorithm in detail.

5.2.3 Task Grouping and Data Locality Inference

Algorithm 6: Task grouping

```

Func GroupTasks ( $T$ ) :
1   $G \leftarrow \{\}$ 
2   $\theta \leftarrow \emptyset$ 
3  for  $\tau \in T$  do
4      if  $\tau.dataPlacement \neq \emptyset$  then
5           $\theta \leftarrow \tau.dataPlacement$ 
6      else if  $\tau.dependencies \neq \emptyset$  then
7           $\theta \leftarrow \text{Host } \eta \text{ where most of } \tau\text{'s predecessors are placed}$ 
8      else if  $\tau.application.anchor \neq \emptyset$  then
9           $\theta \leftarrow \tau.application.anchor$ 
10     else
11          $\theta \leftarrow \text{Random host}$ 
12          $\tau.application.anchor \leftarrow \theta$ 
13     if  $g_\theta = \emptyset$  then
14          $g_\theta \leftarrow \{\}$ 
15      $\tau.anchor \leftarrow \theta$ 
16      $g_\theta \leftarrow g_\theta \cup \{\tau\}$ 
17      $G \leftarrow G \cup \{g_\theta\}$ 
18 return  $G$ 

```

The algorithm first groups the *tasks* by the location of their input data if any, i.e., the *anchor* of the *tasks* by definition. In effect, each task group is formed centering around an *anchor*. However, finding the *anchor* for a *task* is difficult when the location of input data is not explicitly given – it is common among *tasks* performing ad-hoc data transfers with each other instead of using the *unified storage*, in which the locations of *volumes* are visible to the algorithm.

To handle this case, we have developed the *data locality inference* approach to infer the *anchor* of a task from implicit information embedded in its specification. Specifically, we primarily use *application locality*

and task dependencies as evidences. Intuitively, we assume that *tasks* belonging to the same *application* are likely to have data exchanges for inter-operations and communication; therefore, co-locating these *tasks* is conducive to retaining the data locality. Furthermore, task dependencies implicitly indicate the data flow among *tasks* – they tend to consume output data from their predecessors and produce intermediate data for their descendants; therefore, placing *tasks* with dependencies among each other may improve the data locality.

Algorithm 6 shows the logic of task grouping phase of the algorithm. If the location of the input data is explicitly given, the algorithm will directly use it as the *anchor* (Line 4–5); otherwise, it takes task dependencies (Line 5–6) in priority over the application locality (Line 6–7) as the evidence for finding the *anchor*, since we consider the former as a stronger signal indicating that data transfers are likely to occur between *tasks*. If neither evidence is available, which is common for the initial *task* of an *application*, the algorithm will randomly select an *anchor* for the *task* and the *application* as well, such that peer *tasks* will be scheduled accordingly to the application locality (Line 7–9).

5.2.4 Task Ordering

With the *tasks* grouped up properly, the algorithm starts the task placement iteratively for each group. Essentially, it adopts the first fit decreasing (FFD) algorithm to each group by sorting the *tasks* in the decreasing order of their resource demand, since FFD is able to pack the *tasks* into fewer VM instances than FF (Dósa, 2007), therefore reducing the VM subscription cost.

However, different from the size of an *item*, the resource demand of a *task* is a 4-dimensional vector as introduced in Section 5.2.1, which cannot be compared in the one-dimensional space. To reduce the dimensionality, we calculate the L_2 -norm of the resource demand vector, i.e., $\|d_\tau\|_2$, as the "size" of a *task*. Algorithm 5.2.4 illustrates the comparator function for comparing the resource demand between *tasks*. This

dimension reduction approach is identified effective in the realm of VBP as revealed in (Panigrahy et al., 2011).

Algorithm 7: Comparator function for task ordering

Func CompareTask (τ_α, τ_β) :

```

1   $d_{\tau_\alpha} \leftarrow \tau_\alpha.cpus^2 + \tau_\alpha.ram^2 + \tau_\alpha.disk^2 + \tau_\alpha.gpus^2$ 
2   $d_{\tau_\beta} \leftarrow \tau_\beta.cpus^2 + \tau_\beta.ram^2 + \tau_\beta.disk^2 + \tau_\beta.gpus^2$ 
3  if  $d_{\tau_\alpha} = d_{\tau_\beta}$  then
4    return 0
5  if  $d_{\tau_\alpha} > d_{\tau_\beta}$  then
6    return -1
7  return 1

```

5.2.5 VM Ordering

Unlike *bins* in the classic VBP, VM instances are distinct in several aspects and from the perspective of different *tasks*. Hence, the ordering of VM instances makes a critical impact on the effectiveness of the scheduling algorithm. Generally, the algorithm tends to place a *task* onto a VM instance with the most resources but the least cost. To capture these factors quantitatively, we introduce a function as below to give scores to VM instances, which indicate their priority in task placement, i.e., VM instances with higher scores will be used for placing *tasks* in prior to those with lower scores.

$$Score(\eta, \theta) = \frac{\|R_\eta\|_2 \cdot (b_{\eta\theta} + b_{\theta\eta})}{C_{\eta\theta} + C_{\theta\eta} + \epsilon} \quad (5.6)$$

In this function, we factor in bidirectional network bandwidth (b) and egress cost (C) between the host η and the *anchor* (θ). The score is positively correlated with the network bandwidth but inversely with the egress network traffic cost from and to the *anchor*. Further, we use the L_2 -norm of the resource capacity vector, i.e., $\|R_\eta\|_2$, to capture the resource availability of a VM instance in the one-dimensional space, which is also positively correlated with the score. The algorithm sorts the VM instances in the decreasing order

of their scores, such that *tasks* can land on high-score VM instances first. Algorithm 5.2.5 describes the comparator function for sorting the VM instances.

Algorithm 8: Comparator function for sorting VM instances

Func CompareVM($\eta_\alpha, \eta_\beta, \theta$) :

```

1   $s_{\eta_\alpha} \leftarrow \frac{\|R_{\eta_\alpha}\|_2 \cdot (b_{\eta_\alpha \theta} + b_{\theta \eta_\alpha})}{C_{\eta_\alpha \theta} + C_{\theta \eta_\alpha} + \epsilon}$ 
2   $s_{\eta_\beta} \leftarrow \frac{\|R_{\eta_\beta}\|_2 \cdot (b_{\eta_\beta \theta} + b_{\theta \eta_\beta})}{C_{\eta_\beta \theta} + C_{\theta \eta_\beta} + \epsilon}$ 
3  if  $s_{\eta_\alpha} = s_{\eta_\beta}$  then
4  |   return 0
5  if  $s_{\eta_\alpha} > s_{\eta_\beta}$  then
6  |   return -1
7  return 1

```

5.2.6 First-Fit Vector Bin Packing

After sorting the *tasks* in the group and VM instances accordingly, the algorithm uses the FF algorithm to fit the *tasks* into the VM instances as shown in Algorithm 9. It iterates over the *tasks* and VM instances in their given order (Line 1–2) and makes the task placement when finding the first fit (Line 3–4); lastly, it deducts the resources allocated to the *task* from the VM instance where it is placed (Line 5–8).

Algorithm 9: First-Fit algorithm

Func FirstFit($g, hosts$) :

```

1  for  $\tau \in g$  do
2  |   for  $\eta \in hosts$  do
3  | |   if  $\tau.cpus \leq \eta.cpus$  and  $\tau.ram \leq \eta.ram$  and  $\tau.disk \leq \eta.disk$  and  $\tau.gpus \leq \eta.gpus$ 
4  | | |   then
5  | | | |    $\tau.placement \leftarrow \eta$ 
6  | | | |    $\eta.cpus \leftarrow \eta.cpus - \tau.cpus$ 
7  | | | |    $\eta.ram \leftarrow \eta.ram - \tau.ram$ 
8  | | | |    $\eta.disk \leftarrow \eta.disk - \tau.disk$ 
9  | | | |    $\eta.gpus \leftarrow \eta.gpus - \tau.gpus$ 

```

5.3 Evaluation

To evaluate the effectiveness and feasibility of our approach in PIVOT, we deploy PIVOT across AWS and GCP. We drive the experiments with simulations on Alibaba production cluster trace and real-world big data applications to investigate the system and proposed algorithm in depth. For performance metrics, we focus on the cost savings in VM subscription and egress network traffic, but also observe the efficiency of application executions and data transfers.

5.3.1 Experiment Setup

5.3.1.1 Alibaba Cluster Trace and Simulation

The Alibaba cluster trace provides an 8-day collection of batch job trace in their data centers in 2018. Each job consists of a number of tasks with data dependencies among them specified in the data set. In this evaluation, we randomly sample a total of 35,000 batch jobs (3,181,620 tasks) from the trace data set. Each job is run as an application of PIVOT respecting the data dependencies among the tasks. Each task produces and transfers 10 – 400MB data. The data size is proportional to the RAM demand of each task. We simulate the PIVOT deployment on 600 VM instances evenly distributed among 31 AZs of the 11 North America regions on AWS and GCP as shown in Figure 5.3. Table 5.1 reflects the egress network traffic among cloud regions in AWS and GCP as of the evaluation. Each VM instances is configured with 16 CPUs and 128GB RAM (*r5.4xlarge* and alike instance/machine type). The network is constructed based on a two-week network trace collection among regions and clouds in AWS and GCP using *iperf3*. The simulator is written in Python using Simpy and available on Github¹².

Cloud Vendor	Traffic Type	Cost
AWS	us-east-1 ↔ us-east-2	0.01
	Between AWS regions	0.02
	To GCP regions	0.09
GCP	Between GCP regions	0.01
	To AWS regions	0.11

Table 5.1: Egress network traffic cost within and between AWS and GCP

¹²PIVOT Scheduling Simulator: <https://github.com/heliumdatacommons/pivot-scheduling>

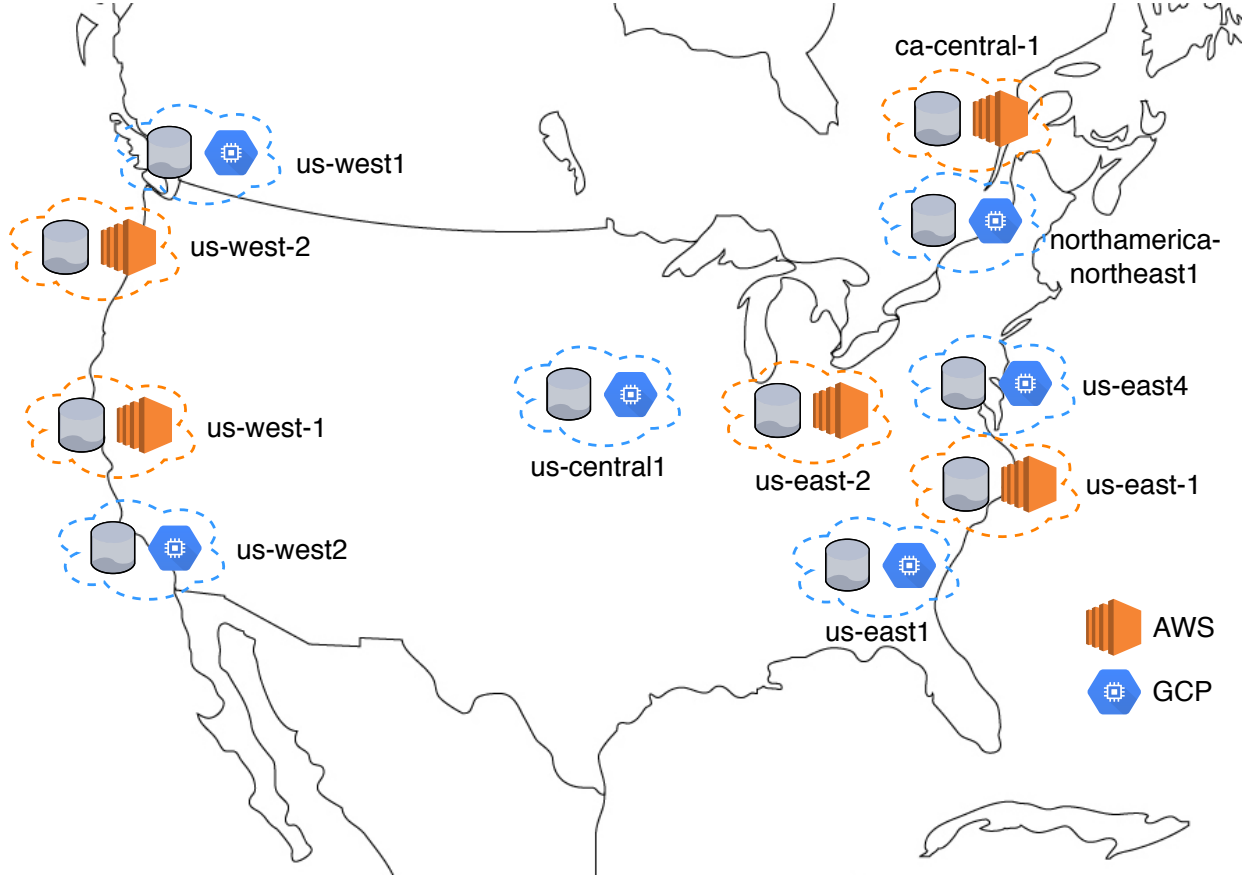


Figure 5.3: Cloud regions in AWS and GCP used for evaluating PIVOT

5.3.1.2 Big Data Applications and Real Deployment

We also replicate the PIVOT deployment in the real world to evaluate the system and algorithm in depth with big data applications. Different from the simulation, each VM is configured with 4 Skylake CPUs, 8GB RAM and 150GB disk space (*c5.xlarge* and alike). In this deployment, there are 100 VM instances evenly distributed across the 31 AZs.

We introduce two featured, real-world use cases - 1) the Hail¹³ genomic analysis and 2) the TOPMed alignment workflow¹⁴. Both workloads exhibit high-level of parallelism, computation intensity and data dependencies thus stressing the challenges encountered in the cloud-distributed environments we consider in this work. Hail is an open-source genomic analytical tool running on Spark¹⁵ to enable large-scale genomic analysis; the TOPMed alignment workflow is an example of workflows encoded in Common Workflow

¹³Hail: <https://github.com/hail-is/hail>

¹⁴DataBiosphere/topmed-workflows: <https://github.com/DataBiosphere/topmed-workflows>

¹⁵Apache Spark: <https://spark.apache.org/>

Language (CWL) (Amstutz et al., 2016) and publicly available at Dockstore¹⁶. The Hail workload are executed as regular Spark applications atop containerized Spark clusters scheduled and run by PIVOT, in which data processing tasks are executed and the intermediate data is exchanged among distributed, containerized *workers*. The TOPMed workflow used for this experiment consists of parallel data processing tasks with dependencies among each other. The number of parallel tasks varies between 10 – 70. The topology of the dependency graph is a MapReduce structure starting with splitting the input, two intermediate phases processing the chunks, and a final aggregation phase. The high level of parallelism and data dependencies lends itself well to analysis of distributed scheduling algorithms. We have ported both Hail cluster and CWL workflows as applications runnable on PIVOT to serve the biomedical community and enable them to take advantage of the cross-cloud scalability.

5.3.1.3 Baseline

In our evaluation, we compare our cost-aware to the following baseline algorithms.

- *Opportunistic* is a common scheduling strategy that assigns tasks to VM instances with sufficient resources *opportunistically* for high resource utilization in overall as adopted in Hindman et al. (2011) and Boutin et al. (2014). In our implementation, the scheduler assigns tasks randomly to the VM instances where they can fit in.
- VBP consists of the FF and BF family of algorithms.
- *Mesos* (Hindman et al., 2011) uses a *resource offer* mechanism that achieves high data locality and scalability within the data center. It is a sophisticated adaptation of the *Opportunistic* algorithm. In the evaluation, we compared our algorithm to *Mesos* with real applications.

5.3.2 Results

5.3.2.1 Simulation on Alibaba Cluster Trace

Figure 5.4 compares our algorithm to the baseline algorithms across several aspects. As shown in the figure, the *Opportunistic* strategy performs the worst in cost efficiency since it does neither task packing nor is cost aware. This follows intuition since this scheduling strategy tends to spread out tasks randomly

¹⁶Dockstore: <https://dockstore.org/>

across regions, thus fragmenting resource utilization on the VM instances and leading to high cost in host subscription and egress cost. In contrast, the `cost-aware` strategy saves up to 90.8% and 99.2% of the host subscription and egress cost respectively as compared to the `Opportunistic` algorithm. Although saving comparably 92.1% in host subscription cost due to the effective consolidation of tasks into fewer VM instances, the `VBP` only reduces 15.9% of the egress cost. This is mainly because `VBP` is oblivious to data locality and the egress cost model and blindly scales out applications across regions and clouds, thus causing excessive egress cost. In comparison, by grouping tasks and sorting VM instances respecting data locality, `cost-aware` is able to schedule tasks in proximity to their input data and *radially* scales out applications centered around their *anchor*. Effectively, the algorithm favors cost-efficient VM instances when placing tasks and therefore achieves the greatest cost saving. As the result of the high level of data locality achieved by the algorithm, `cost-aware` incurs the least application runtime on average.

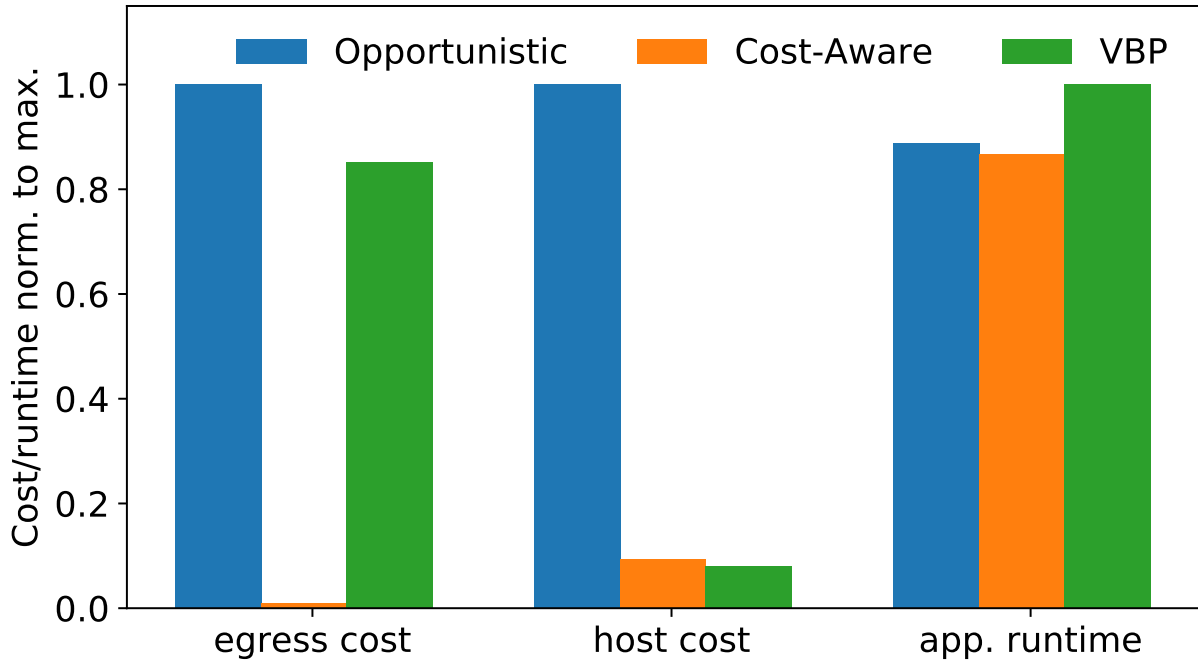


Figure 5.4: Comparison of egress cost, host subscription cost, average number of VM instances used and application runtime. `cost-aware` saves up to 90.8% and 99.2% of the cost for host subscription and egress traffic.

Interestingly, we observe `cost-aware` also achieves the least average application runtime as compared to the baselines. In further analysis, we find that the runtime improvement is mostly due to the reduced data transfer time. As illustrated in Figure 5.5, `cost-aware` saves up to 82.8% of the data transfer time per

task. Notably, VBP performs the worst due to the significant delay caused by network congestion, which represents 85% of the data transfer time. Moreover, we find that the congestion exacerbates when the VM instances are upgraded with more CPUs and RAM. This is because the increase of network bandwidth can rarely keep up with the increase of CPUs and RAM during VM upgrade in the cloud. More concurrently running tasks tend to compete for the limited bandwidth and aggravate the network congestion. Hence, we argue that adopting naive bin-packing strategy can be detrimental to application scheduling in the distributed cloud environment. The result also highlights the importance of the awareness of network bandwidth and data locality for the scheduling algorithm to make wise scheduling decisions in the distributed cloud environment. More specifically, without the awareness of network bandwidth and data locality, VBP can place tasks onto VM instances with bandwidth-limited network paths to their *anchor* and increase the data transmission delay. To worsen the situation, the task packing further stresses the constrained network path and thus creates *hot spots*, where excessive network traffic overwhelms the congested path. As a consequence, VBP greatly slows down the application execution as shown in Figure 5.4. In contrast, since the bandwidth factor outweighs other factors in the host scoring function when the network bandwidth being oversubscribed, the `cost-aware` algorithm is able to avoid the congestion dynamically and trade reasonable resource fragmentation for load balancing in the long run.

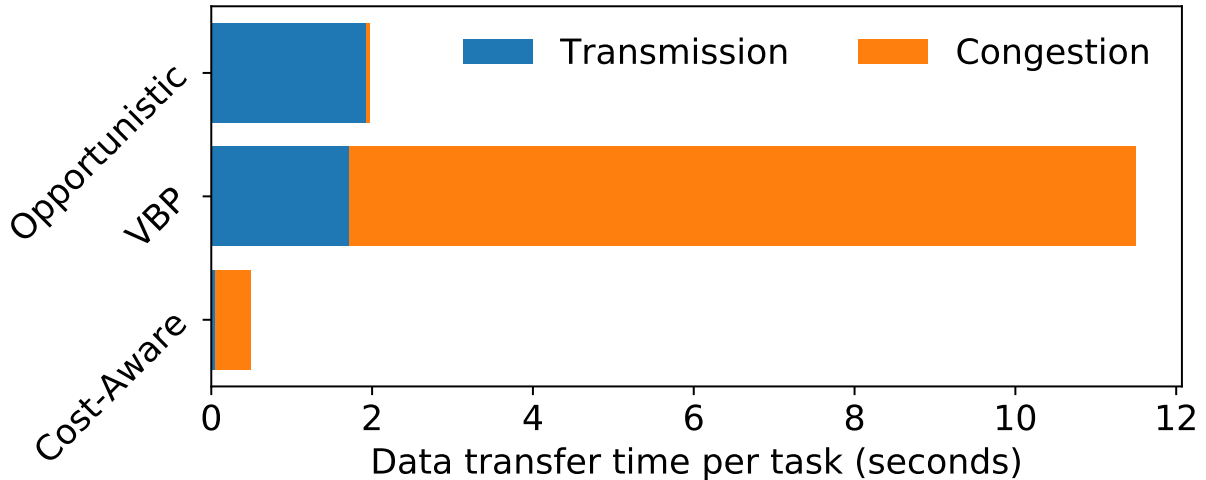


Figure 5.5: Average data transfer time per task. `cost-aware` saves up to 82.8% of data transfer time due to strategic selection of fast network path and avoidance of network congestion.

Figure 5.6 shows the financial cost as a function of the number of applications running in PIVOT. As observed, `cost-aware` incurs trivial egress cost – it incurs at most \$13.62 as compared to >\$1,000 incurred

by the baselines. More importantly, the cost increases at the lowest rate with the increasing system load. The results demonstrate that `cost-aware` is able to limit the footprints of applications within the fewest regions and scales out the applications *strategically* to minimize the egress cost. Furthermore, as the number of applications increases, all the evaluated algorithms inevitably scale out the applications as reflected in the increasing cost for host subscription. However, `cost-aware` exhibits the comparable efficiency in task packing to VBP – it incurs only 16.3% additional host subscription cost on average for improved load balancing and congestion avoidance as shown in Figure 5.5.

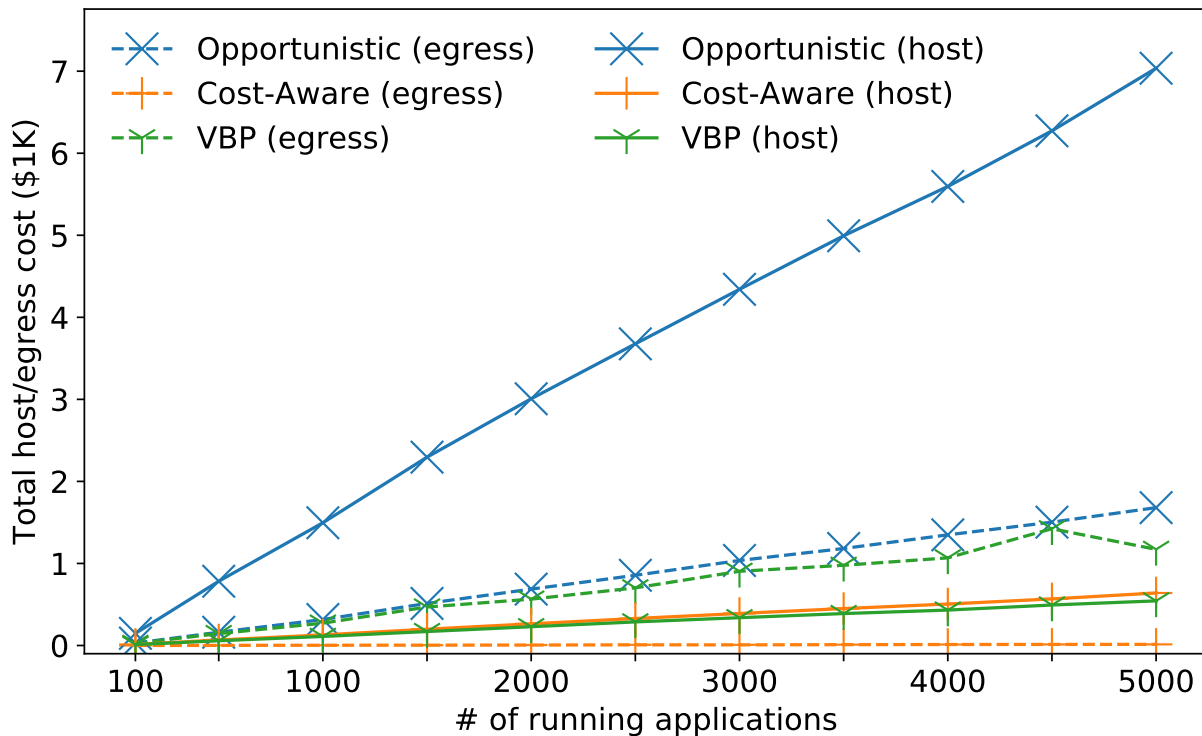


Figure 5.6: Variation of cloud expenses with increasing number of running applications. Despite the lowest egress and host subscription cost, cost-aware achieves mildest cost increase as applications scale up.

5.3.2.2 The Hail Cluster

A Spark cluster consists of a set of *worker processes* (workers) that execute data processing jobs. In PIVOT, Hail *workers* are considered long-running tasks in our workload model. The management of the Hail workload is managed by the Spark framework and therefore is completely *opaque* to PIVOT. Therefore, our cost-aware algorithm only takes into account information about the application locality, *i.e.*, all Hail tasks (workers) are grouped, when making scheduling decisions. To compare the effectiveness of our algorithm

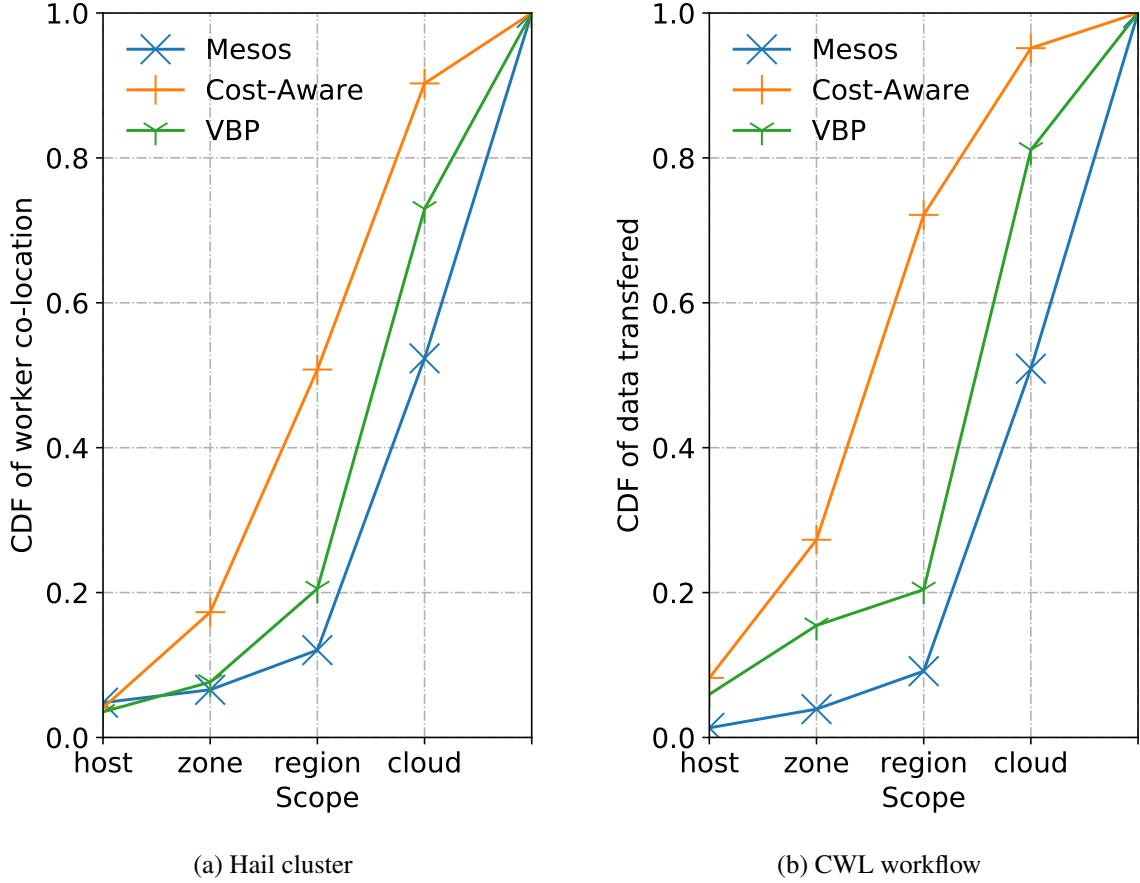


Figure 5.7: CDF of worker co-locations/data transferred for Hail clusters/CWL workflows in different scopes, respectively. cost-aware effectively co-locates the workers/tasks for improved data locality and less egress cost.

against VBP and Mesos, we launch Hail clusters with 10 – 100 *workers*; following we discuss the results of this experiment.

Figure 5.7(a) illustrates the placement distribution of *workers* for Hail clusters of varying sizes. As observed, the *cost-aware* algorithm co-locates the majority of the *workers* within the same region and cloud (50.8% and 90.3%, respectively).

Additionally, in Figure 5.8 we show a cost-throughput trade off analysis for all three scheduling algorithms. Notice that the origin of the graph represents the optimum point at which data transfers can be completed instantly without any monetary cost, and every point represents the average data transfer throughput and egress cost for a Hail cluster instance. As observed, most scheduling decisions under the *cost-aware* algorithm are close to the optimum while those under Mesos and VBP are concentrated on the top right corner of the graph. This follows intuition since by co-locating *workers* the *cost-aware*

algorithm improves data locality (as reflected in throughput) and reduces egress cost. This is in contrast to the low throughput resulting from the sparse distribution of *workers* when using Mesos and VBP. Note the outlier placement decisions landing at the far top right corner under the *cost-aware* scheduler. These outliers reveal the limitation of application-locality-based scheduling as application-locality does not imply data-locality in a distributed system (an assertion valid in centralized environments, e.g., single node). We argue that considering the limited information provided by the application to the scheduler, *cost-aware* strikes a good trade-off between cost and data locality improvement.

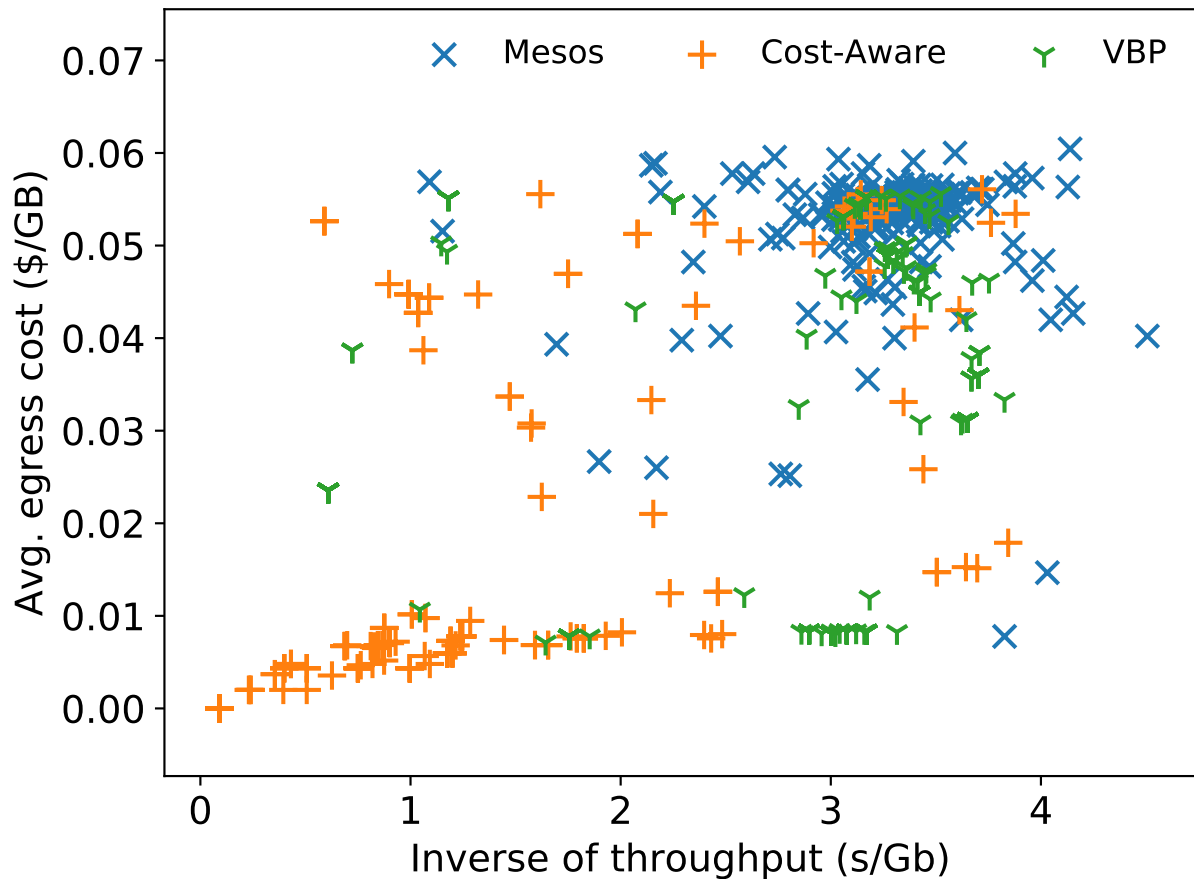


Figure 5.8: Footprints of the Hail cluster deployments in the cost-throughput space. Most deployments by cost-aware are clustered in proximity to the optimum, while those by the baselines mostly distributed at the far end.

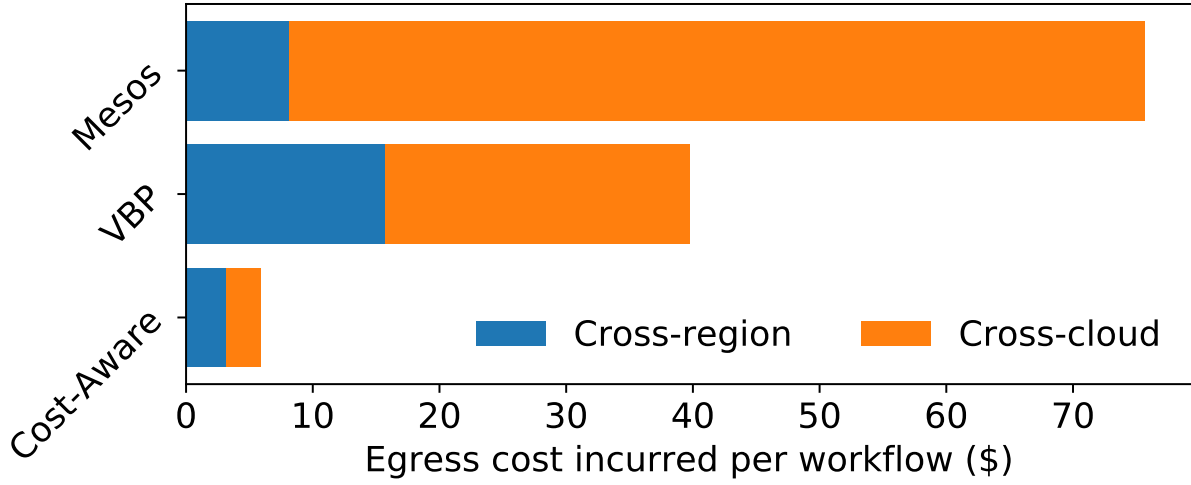


Figure 5.9: Average egress cost incurred by running the CWL workflows. *cost-aware* saves up to \$2,092 (92.2%) in total for 30 workflow runs

5.3.2.3 CWL Workflows

CWL workflows are representative of the applications considered in our earlier simulation analysis. More specifically, data dependencies among workflow jobs are defined in the CWL workflow description¹⁷ – described as input and output files for each step of the workflow. Thus, the scheduling algorithm can take advantage of this additional information to infer data locality accurately when making placement decisions.

In Figure 5.7, we notice that the locality of interdependent tasks which read their input data from a preceding task increases significantly under the *cost-aware* scheduler. A more detailed analysis of our results shows that once the anchor tasks are placed, the scheduling strategy makes a best effort to balance locality and performance as reflected by the *clustering* of descendent tasks.

Recall that cross-region cost is significant, therefore we are concerned with keeping data transfers within cloud regions. Note that in Figure 5.7, 9.1%, 20.4%, and 72.1% of data transfers were kept within a region using the *Mesos*, *VBP*, and *cost-aware* schedulers, respectively. The steep CDF slope of the *cost-aware* strategy demonstrates the effectiveness of our algorithm; only 4.9% of transfers span clouds compared to 18.9% and 49.0% for *VBP* and *Mesos*, respectively. Turning to Figure 5.9 we see a drastic reduction in egress charges, and a shift from charges mostly being from cross-cloud in *Mesos* (89.2%) and *VBP* (60.2%), to charges being mostly from cross-region within a cloud in *cost-aware* (53.4%).

¹⁷CWL Workflow Description, v1.0.2: <https://www.commonwl.org/v1.0/Workflow.html>

5.4 Chapter Summary

In this chapter, we have introduced PIVOT, a cloud-agnostic computing platform for enabling execution and scaling of data-intensive applications across geo-distributed cloud regions and platforms. Specifically, we elaborate on the resource abstraction mechanism that overcomes the technical and business barriers between cloud regions and platforms, which facilitates the seamless application scaling across cloud. Additionally, we have devised a two-level scheduling framework that allows the co-existence of applications with various scheduling goals. Furthermore, in recognition of the substantial cloud expenses incurred by running geo-distributed, data-intensive applications across cloud, we have proposed a cost-aware scheduling algorithm that optimally places applications to minimize the cloud expenses and improves application performance in a best-effort manner. The experimental evaluation indicates that PIVOT is able to host large-scale, data-intensive applications composed of hundreds of parallel computing tasks; the cost-aware scheduling algorithm also achieves significant cost saving and performance improvement as compared to the baselines.

CHAPTER 6: Conclusion and Future Work

Data-intensive applications are commonly run in the geo-distributed environment since numerous masses of data are generated and stored at widely distributed geo-locations, and their efficiency is of the utmost importance. Although significant work has been done to improve efficiency of data-intensive applications within data centers, there is an inadequacy of literature on scaling these applications efficiently in the geo-distributed environment. The main objective of this dissertation is to identify and tackle core challenges in ensuring efficiency of data-intensive applications in the geo-distributed environments. Towards this goal, we take a combined approach of system analysis, design and engineering to clarify and address the key challenges of resource heterogeneity, data locality and network efficiency.

This chapter is organized as follows. In Section 6.1, we provide a brief summary of the results made in this dissertation. In Section 6.2, we briefly introduce work done by the author during his doctoral study in addition to those included in this dissertation. In Section 6.3, we propose potential future work related to this dissertation.

6.1 Summary of Results

Centering on the key challenges identified in the thesis statement, the results presented in this dissertation can be summarized as follows.

Cross-layer resource abstraction for easing heterogeneity. We have performed resource abstraction throughout a geo-distributed system across multiple layers to mitigate the heterogeneity in geo-distributed environments and simplify the procedures for users and services to use the system. In the meantime, the cross-layer resource abstraction serves as the foundation for the systematic and algorithmic optimizations on top. Specifically, in Chapter 3, we introduce a DFD-based model at the API level to inclusively represent complex, data-centric scientific collaboration, simplifying the usage of virtualized geo-distributed infrastructure for domain scientists without technical backgrounds in scientific research. In addition, the SDN-based WAN abstraction at the infrastructure level creates the basis for the global WAN optimization for remote bulk data

transfers. In Chapter 4, we introduce a middleware-level abstraction for reusable tasks that can be cached in a geo-distributed cache network, guiding the implementation of reusable tasks and encapsulation mechanism for such tasks. Meanwhile, we also define the interface for the underlying geo-distributed caching system at the infrastructure level. In Chapter 5, we abstract computing and storage provisioned by multiple cloud providers at infrastructure and middleware levels to achieve cloud agnosticism, therefore enabling the flexible cost-aware task scheduling on top. In a nutshell, these cross-layer resource abstraction approaches can be adopted partially or collectively in a geo-distributed system to facilitate data-intensive applications.

Network-aware mechanisms for improved data locality. We recognize the importance of data locality for data-intensive applications, especially in geo-distributed environments, as indicated by a bulk of literature presented in Chapter 2. Further, we realize the inadequacy of research on the impact of networks on decision making pertaining to preserving data locality in the geo-distributed environment – prior work mostly focuses on data center environments wherein networks are homogeneous and efficient, which contrasts with variable, unpredictable WANs in geo-distributed environments. Hence, we enhance network awareness in our approaches to improving data locality for geo-distributed, data-intensive applications. In Chapter 3, the proposed optimization algorithm favors network paths with lower latency between two ends of a data transfer. In Chapter 4, we factor bandwidth availability among cache servers into the decision making for cache replacement and replication to ensure data locality for high-value, frequently-repeated tasks. The experimental evaluation reveals the enormous impact of network awareness on cache hit rate and saving in job completion time. In Chapter 5, the global task scheduling algorithm also takes into account real-time bandwidth measurements in addition to cost factors for prioritizing hosts during task assignment, effectively placing tasks in proximity to their input data for data locality and cost saving. The experiment also reflects that the network awareness contributes to the reduction of end-to-end runtime of data-intensive applications.

WAN optimization for remote bulk data transfers. We also contribute to WAN optimization for performance improvement in remote bulk data transfers, which are predominantly performed in geo-distributed, data-intensive applications, since it resolves low-level performance bottlenecks at the root that can hardly be circumvented through high-level mechanisms. Specifically, in Chapter 3, we developed a combined network optimization solution comprising MCF-based global optimization and TCP enhancement on top of the network abstraction, which optimizes bandwidth allocation and path assignment to individual data transfers based on their specified priority levels and ensures the maximal utilization of the allocated bandwidth. The experimental evaluation shows significant improvement in bandwidth utilization in WANs, which eventually

benefit remote data transfers as they gain speedup from increasing usable bandwidth in the networks. Further, this work also exemplifies the success of resource abstraction in eliminating the heterogeneity in WANs, since such optimization is hardly possible in traditional heterogeneous WANs.

6.2 Other Work

The following is a brief summary of other work done by the author in parallel with this dissertation during his doctoral study.

Enabling workflow repeatability with virtualization support.¹ Repeatability of scientific workflows is crucial for scientific research, since workflows often need to be replayed over time in varying environments to reproduce and re-validate results previously generated by them. However, it is challenging to not only reproduce results but also achieve consistent performance and resource consumption in varying time and space. Hence, this work focuses on developing a comprehensive abstraction, referred to as workflow virtual appliance (WVA), which encapsulates every aspect of a scientific workflow from infrastructure to application with virtualization support to achieve the workflow repeatability. Specifically, it relies on the IaaS service to capture the resource subscription in the underlying infrastructure. On top of that, the WVA abstracts various resources using HTCondor (Litzkow et al., 1987) and exposes a unified interface to the workflows for job scheduling among the distributed resources. For the workflow, Pegasus (Deelman et al., 2016) is adopted as the standard workflow framework to abstract a DAG-based workflow, which is representative for the majority of scientific workflows. A middleware named WRAP is developed to orchestrate the various components within a WVA and across WVA instances. As a result of the experimental evaluation, the system achieves strongly consistent repeatability for the Montage (Bharathi et al., 2008) and exomic alignment workflows. This work is a driving case for RADII, in which we extend the abstraction to general data-centric scientific applications beyond solely workflows.

6.3 Future Work

Chunk-based data caching and deduplication. In Chapter 4, we elaborate on a file-based caching mechanism and algorithm for computation caching, in which the atomic unit of caching is a file that contains

¹Details of this contribution have been published in the following paper:

F. Jiang, C. Castillo, C. Schmitt, A. Mandal, P. Ruth, and I. Baldin, “Enabling workflow repeatability with virtualization support,” Proceedings of the 10th Workshop on Workflows in Support of Large-Scale Science - WORKS ’15, pp. 1–10, 2015.

the intermediate or output data of a task. The major problem with the file-based caching is that files have heterogeneous sizes and therefore cannot well fit in a cache, causing space fragmentation and wasting valuable cache space. The potential solution is to use chunk-based caching (Cho et al., 2012b; Lim et al., 2014), in which data is stored in the unit of uni-sized chunks that can fully utilize the cache space and therefore avoid space fragmentation. Moreover, chunk-based caching can further save cache space through fine-grained data deduplication (Bhagwat, Deepavali and Eshghi, Kave and Long, Darrell DE and Lillibridge, Mark, 2009; Lillibridge et al., 2013), since it has a higher probability that there are chunks sharing identical contents, in which case only a single copy needs to be cached. However, there are two challenges to overcome in order to realize chunk-based caching at the geo-distributed scale. First, it takes extra time and computing power to marshal/unmarshal data from/to chunks, introducing additional delay for cache insertion and retrieval. Second, it may incur significant overhead to index and orchestrate geo-distributed chunks. With the chunk-based caching, it is expected to observe higher utilization rate of cache space.

Container-level network abstraction. We introduce a VM-level network abstraction in Chapter 3 and a container-based computing system in Chapter 5. However, although being functional, it is inadequate to apply the VM-level network abstraction to the container-based computing system due to the mismatch in granularity of control and consequently complicates the system design. With the popularity of micro-service architecture and container network interface (CNI)², the network abstraction can be enhanced with finer granularity of control over applications – instead of creating a single virtual network shared among multiple applications running on the same VM cluster, each application can have a dedicated virtual network that connects solely its own distributed, containerized services. The fine-grained network abstraction approach can make a profound impact on many aspects. First, it creates better isolation among applications with simplicity since they communicate within their own isolated virtual networks. Second, it reduces the network complexity perceived by the SDN controller, since every application has a dedicated SDN controller which only oversees components associated with the specific application. Last, the network abstraction is more lightweight than the VM-level approach – a software switch can run as a container rather than fully occupy a VM; definition and configuration of a network are mostly logical and software-based, and therefore can be automated. The new network abstraction may create opportunities for innovations in networking for micro-service architecture, which is an area not yet thoroughly investigated.

²CNI: <https://github.com/containernetworking/cni>

Layer-based container image cache management. With a container-based computing system as introduced in Chapter 5, downloading of container images may contribute a major portion of data ingestion of the system. Despite the large size of container images, these images share a significant amount of identical contents that are redundantly downloaded, causing waste of network bandwidth and storage space and delaying application start-up (Anwar et al., 2018; Zheng et al., 2018). Even worse, although a container image implements a union file system built on top of multiple data layers, the state-of-the-art container daemon, namely Docker, only allows full image deletion upon space shortage, lowering the average space utilization rate and forcing duplicate image downloading. Further, it seldom implements any logic for automatic image management but burdens users to make decisions on image deletion, which is often shortsighted or even blind, further aggravating the problem. Hence, it is compelling to develop a system that automatically caches popular data layers of container images to avoid duplicate image downloading and speed up application start-up.

Use of preemptible resources across cloud. Preemptible resources typically have a bargain price on most cloud platforms for the risk of being preempted. Taking advantage of preemptible resources in the cloud can potentially achieve substantial cost saving. For instance, by using preemptible instances on GCP, we manage to cut the cost for a genomic analysis by over 80%. There is a plethora of literature leveraging preemptible resources offered by a single cloud provider to save cloud expenses (Chohan et al., 2010; Qu et al., 2016; Pucher et al., 2017; Tordini et al., 2018), which limits applications from scaling out across cloud and taking advantage of price discrepancies among cloud providers. Hence, it is intriguing to enable use of preemptible resources across cloud while satisfying SLAs of applications, which potentially yields extra cost saving while improving application reliability.

BIBLIOGRAPHY

- Agrawal, P., Kifer, D., and Olston, C. (2008). Scheduling Shared Scans of Large Data Files. *Proceedings of the VLDB Endowment*, 1(1):958–969.
- Ahmad, R. W., Gani, A., Hamid, S. H. A., Shiraz, M., Yousafzai, A., and Xia, F. (2015). A Survey on Virtual Machine Migration and Server Consolidation Frameworks for Cloud Data Centers. *Journal of Network and Computer Applications*, 52:11–25.
- Al-Fares, M., Radhakrishnan, S., Raghavan, B., Huang, N., and Vahdat, A. (2010). Hedera: dynamic flow scheduling for data center networks. In *Nsdi*, volume 10.
- Alizadeh, M., Edsall, T., Dharmapurikar, S., Vaidyanathan, R., Chu, K., Fingerhut, A., Lam, V. T., Matus, F., Pan, R., Yadav, N., et al. (2014). CONGA: Distributed Congestion-Aware Load Balancing for Datacenters. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, pages 503–514.
- Amdahl, G. M. (1967). Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities. In *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, pages 483–485.
- Amstutz, P., Crusoe, M. R., Tijanić, N., Chapman, B., Chilton, J., Heuer, M., Kartashov, A., Leeher, D., Ménager, H., Nedeljkovich, M., Scales, M., Soiland-Reyes, S., and Stojanovic, L. (2016). Common Workflow Language, v1.0.
- Anwar, A., Mohamed, M., Tarasov, V., Little, M., Rupprecht, L., Cheng, Y., Zhao, N., Skourtis, D., Warke, A. S., Ludwig, H., et al. (2018). Improving Docker Registry Design Based on Production Workload Analysis. In *16th USENIX Conference on File and Storage Technologies (FAST 18)*, pages 265–278.
- Assunção, M. D., Calheiros, R. N., Bianchi, S., Netto, M. A., and Buyya, R. (2015). Big Data Computing and Clouds: Trends and Future Directions. *Journal of Parallel and Distributed Computing*, 79:3–15.
- Baldine, I., Xin, Y., Mandal, A., Ruth, P., Heerman, C., and Chase, J. (2012). Exogeni: A Multi-Domain Infrastructure-as-a-Service Testbed. In *Testbeds and Research Infrastructure. Development of Networks and Communities*, pages 97–113. Springer.
- Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I., and Warfield, A. (2003). Xen and the art of virtualization. *ACM SIGOPS operating systems review*, 37(5):164–177.
- Bellard, F. (2005). QEMU, a Fast and Portable Dynamic Translator. In *USENIX Annual Technical Conference, FREENIX Track*, volume 41, page 46.
- Beloglazov, A. and Buyya, R. (2010). Adaptive threshold-based approach for energy-efficient consolidation of virtual machines in cloud data centers. In *MGC@ Middleware*, page 4.
- Beloglazov, A. and Buyya, R. (2012). Optimal Online Deterministic Algorithms and Adaptive Heuristics for Energy and Performance Efficient Dynamic Consolidation of Virtual Machines in Cloud Data Centers. *Concurrency and Computation: Practice and Experience*, 24(13):1397–1420.
- Berman, M., Chase, J. S., Landweber, L., Nakao, A., Ott, M., Raychaudhuri, D., Ricci, R., and Seskar, I. (2014). GENI: A Federated Testbed for Innovative Network Experiments. *Computer Networks*, 61:5 – 23. Special issue on Future Internet Testbeds â Part I.

- Bhagwat, Deepavali and Eshghi, Kave and Long, Darrell DE and Lillibridge, Mark (2009). Extreme Binning: Scalable, Parallel Deduplication for Chunk-Based File Backup. In *2009 IEEE International Symposium on Modeling, Analysis & Simulation of Computer and Telecommunication Systems*, pages 1–9. IEEE.
- Bharathi, S., Chervenak, A., Deelman, E., Mehta, G., Su, M.-H., and Vahi, K. (2008). Characterization of Scientific Workflows. In *2008 Third Workshop on Workflows in Support of Large-Scale Science*, pages 1–10. IEEE.
- Bittencourt, L. F. and Madeira, E. R. M. (2011). HCOC: a Cost Optimization Algorithm for Workflow Scheduling in Hybrid Clouds. *Journal of Internet Services and Applications*, 2(3):207–227.
- Borst, S., Gupta, V., and Walid, A. (2010). Distributed Caching Algorithms for Content Distribution Networks. In *2010 Proceedings IEEE INFOCOM*, pages 1–9.
- Boutin, E., Ekanayake, J., Lin, W., Shi, B., Zhou, J., Qian, Z., Wu, M., and Zhou, L. (2014). Apollo: Scalable and Coordinated Scheduling for Cloud-Scale Computing. In *OSDI*, volume 14, pages 285–300.
- Brakmo, L. S. and Peterson, L. L. (1995). TCP Vegas: End to End Congestion Avoidance on a Global Internet. *IEEE Journal on Selected Areas in Communications*, 13(8):1465–1480.
- Braun, W. and Menth, M. (2014). Software-Defined Networking Using OpenFlow: Protocols, Applications and Architectural Design Choices. *Future Internet*, 6(2):302–336.
- Breslau, L., Cao, P., Fan, L., Phillips, G., and Shenker, S. (1999). Web Caching and Zipf-Like Distributions: Evidence and Implications. In *IEEE INFOCOM '99. Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings*, volume 1, pages 126–134 vol.1.
- Broder, A. Z. (1993). Some Applications of Rabin’s Fingerprinting Method. In *Sequences II: Methods in Communications, Security, and Computer Science*, pages 143–152. Springer-Verlag.
- Buyya, R., Ranjan, R., and Calheiros, R. N. (2010). Intercloud: Utility-Oriented Federation of Cloud Computing Environments for Scaling of Application Services. In *International Conference on Algorithms and Architectures for Parallel Processing*, pages 13–31. Springer.
- Buyya, R., Yeo, C. S., Venugopal, S., Broberg, J., and Brandic, I. (2009). Cloud Computing and Emerging IT Platforms: Vision, Hype, and Reality for Delivering Computing as the 5th Utility. *Future Generation Computer Systems*, 25(6):599–616.
- Cai, C. X., Le, F., Sun, X., Xie, G. G., Jamjoom, H., and Campbell, R. H. (2016). CRONets: Cloud-Routed Overlay Networks. In *2016 IEEE 36th International Conference on Distributed Computing Systems (ICDCS)*, pages 67–77.
- Calder, M., Fan, X., Hu, Z., Katz-Bassett, E., Heidemann, J., and Govindan, R. (2013). Mapping the Expansion of Google’s Serving Infrastructure. In *Proceedings of the 2013 Conference on Internet Measurement Conference*, pages 313–326.
- Caneill, M., El Rheddane, A., Leroy, V., and De Palma, N. (2016). Locality-Aware Routing in Stateful Streaming Applications. In *Proceedings of the 17th International Middleware Conference, Middleware '16*, pages 4:1–4:13, New York, NY, USA. ACM. event-place: Trento, Italy.
- Cao, J., Zhang, Y., Cao, G., and Xie, L. (2007). Data Consistency for Cooperative Caching in Mobile Environments. *Computer*, 40(4).

- Cao, P. and Irani, S. (1997). Cost-Aware WWW Proxy Caching Algorithms. In *USENIX Symposium on Internet Technologies and Systems*, volume 12, pages 193–206.
- Cardwell, N., Cheng, Y., Gunn, C. S., Yeganeh, S. H., and Jacobson, V. (2016). BBR: Congestion-Based Congestion Control. *Queue*, 14(5):20–53.
- Casado, M., Freedman, M. J., Pettit, J., Luo, J., McKeown, N., and Shenker, S. (2007). Ethane: Taking control of the enterprise. *ACM SIGCOMM computer communication review*, 37(4):1–12.
- Chambers, C., Raniwala, A., Perry, F., Adams, S., Henry, R. R., Bradshaw, R., and Weizenbaum, N. (2010). FlumeJava: Easy, Efficient Data-Parallel Pipelines. In *ACM Sigplan Notices*, volume 45, pages 363–375. ACM.
- Chand, N., Joshi, R. C., and Misra, M. (2007). Cooperative Caching Strategy in Mobile Ad Hoc Networks Based on Clusters. *Wireless Personal Communications*, 43(1):41–63.
- Chekuri, C. and Khanna, S. (1999). On multi-dimensional packing problems. In *Proceedings of the tenth annual ACM-SIAM symposium on Discrete algorithms*, pages 185–194. Society for Industrial and Applied Mathematics.
- Chen, Y., Alspaugh, S., and Katz, R. (2012a). Interactive Analytical Processing in Big Data Systems: A Cross-Industry Study of MapReduce Workloads. *Proceedings of the VLDB Endowment*, 5(12):1802–1813.
- Chen, Y., Grifflit, R., Zats, D., and Katz, R. H. (2012b). Understanding TCP Incast and Its Implications for Big Data Workloads. *University of California at Berkeley, Tech. Rep.*
- Chervenak, A., Deelman, E., Foster, I., Guy, L., Hoschek, W., Iamnitchi, A., Kesselman, C., Kunszt, P., Ripeanu, M., Schwartzkopf, B., and others (2002). Gigggle: a framework for constructing scalable replica location services. In *Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, pages 1–17. IEEE Computer Society Press.
- Cho, K., Lee, M., Park, K., Kwon, T. T., Choi, Y., and Pack, S. (2012a). Wave: Popularity-Based and Collaborative In-Network Caching for Content-Oriented Networks. In *Computer Communications Workshops (INFOCOM WKSHPS), 2012 IEEE Conference on*, pages 316–321. IEEE.
- Cho, K., Lee, M., Park, K., Kwon, T. T., Choi, Y., and Pack, S. (2012b). WAVE: Popularity-Based and Collaborative In-Network Caching for Content-Oriented Networks. In *2012 Proceedings IEEE INFOCOM Workshops*, pages 316–321. IEEE.
- Chohan, N., Castillo, C., Spreitzer, M., Steinder, M., Tantawi, A. N., and Krintz, C. (2010). See Spot Run: Using Spot Instances for MapReduce Workflows. *HotCloud*, 10:7–7.
- Chung, A., Park, J. W., and Ganger, G. R. (2018). Stratus: cost-aware container scheduling in the public cloud. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 121–134. ACM.
- Corbett, J. C., Dean, J., Epstein, M., Fikes, A., Frost, C., Furman, J. J., Ghemawat, S., Gubarev, A., Heiser, C., and Hochschild, P. (2013). Spanner: Google’s globally distributed database. *ACM Transactions on Computer Systems (TOCS)*, 31(3):8.
- Corradi, A., Fanelli, M., and Foschini, L. (2014). VM Consolidation: A Real Case Based on OpenStack Cloud. *Future Generation Computer Systems*, 32:118–127.

- Crow (1987). *Lognormal Distributions: Theory and Applications*. CRC Press, New York, 1 edition edition.
- Cukier, K. (2010). Data, data everywhere. <https://www.economist.com/special-report/2010/02/27/data-data-everywhere> (Accessed on February 23, 2020).
- Daemen, J. and Rijmen, V. (2001). Announcing the advanced encryption standard (aes). *Federal Information Processing Standards Publication*, 197.
- Dahlin, M. D., Wang, R. Y., Anderson, T. E., and Patterson, D. A. (1994). Cooperative Caching: Using Remote Client Memory to Improve File System Performance. In *Proceedings of the 1st USENIX Conference on Operating Systems Design and Implementation*, OSDI '94, Berkeley, CA, USA. USENIX Association.
- Dean, J. and Ghemawat, S. (2004). MapReduce: Simplified Data Processing on Large Clusters. In *OSDI'04: Sixth Symposium on Operating System Design and Implementation*, pages 137–150, San Francisco, CA.
- Dean, J. and Ghemawat, S. (2008). MapReduce: Simplified Data Processing on Large Clusters. *Communications of the ACM*, 51(1):107.
- Deelman, E., Vahi, K., Rynge, M., Juve, G., Mayani, R., and da Silva, R. F. (2016). Pegasus in the cloud: Science automation through workflow technologies. *IEEE Internet Computing*, 20(1):70–76.
- Devine, S. W., Bugnion, E., and Rosenblum, M. (2002). Virtualization system including a virtual machine monitor for a computer with a segmented architecture. US Patent 6,397,242.
- Dobre, C. and Xhafa, F. (2014). Intelligent Services for Big Data Science. *Future Generation Computer Systems*, 37:267–281.
- Dragoni, N., Giallorenzo, S., Lafuente, A. L., Mazzara, M., Montesi, F., Mustafin, R., and Safina, L. (2017). Microservices: Yesterday, Today, and Tomorrow. In *Present and Ulterior Software Engineering*, pages 195–216. Springer.
- Dósa, G. (2007). The tight bound of first fit decreasing bin-packing algorithm is $\text{FFD}(I) \leq \frac{11}{9}\text{opt}(I) + \frac{6}{9}$. In *Combinatorics, Algorithms, Probabilistic and Experimental Methodologies*, pages 1–11. Springer.
- Eastlake, D. and Jones, P. (2001). US Secure Hash Algorithm 1 (SHA1). RFC 3174, RFC Editor.
- Floyd, S., Handley, M., Padhye, J., and Widmer, J. (2000). Equation-Based Congestion Control for Unicast Applications. *ACM SIGCOMM Computer Communication Review*, 30(4):43–56.
- Ford, A., Raiciu, C., Handley, M., Bonaventure, O., et al. (2013). TCP Extensions for Multipath Operation with Multiple Addresses. RFC 6824, RFC Editor.
- Fox, G. C. (2011). Data Intensive Applications on Clouds. In *Proceedings of the Second International Workshop on Data Intensive Computing in the Clouds*, pages 1–2.
- Frenk, H., Csirik, J., Labbé, M., and Zhang, S. (1990). On the multidimensional vector bin packing. *University of Szeged. Acta Cybernetica*, pages 361–369.
- Fricker, C., Robert, P., Roberts, J., and Sbihi, N. (2012). Impact of Traffic Mix on Caching Performance in a Content-Centric Network. In *IEEE NOMEN 2012, Workshop on Emerging Design Choices in Name-Oriented Networking*, Orlando, USA.

- Fu, C. P. and Liew, S. C. (2003). TCP Veno: TCP Enhancement for Transmission over Wireless Access Networks. *IEEE Journal on selected areas in communications*, 21(2):216–228.
- Gandomi, A. and Haider, M. (2015). Beyond the Hype: Big Data Concepts, Methods, and Analytics. *International Journal of Information Management*, 35(2):137–144.
- Glassman, S. (1994). A Caching Relay for the World Wide Web. *Computer Networks and ISDN systems*, 27(2):165–173.
- Gleeson, B., Lin, A., Heinanen, J., Armitage, G., and Malis, A. (2000). A Framework for IP Based Virtual Private Networks. RFC 2764, RFC Editor.
- Goldoni, E., Rossi, G., and Torelli, A. (2009). Assolo, a new method for available bandwidth estimation. In *Internet Monitoring and Protection, 2009. ICIMP'09. Fourth International Conference on*, pages 130–136. IEEE.
- Grozev, N. and Buyya, R. (2014). Inter-Cloud Architectures and Application Brokering: Taxonomy and Survey. *Software: Practice and Experience*, 44(3):369–390.
- Gude, N., Koponen, T., Pettit, J., Pfaff, B., Casado, M., McKeown, N., and Shenker, S. (2008). NOX: Towards an Operating System for Networks. *ACM SIGCOMM Computer Communication Review*, 38(3):105–110.
- Gunasekaran, J. R., Thinakaran, P., Kandemir, M. T., Urgaonkar, B., Kesidis, G., and Das, C. (2019). Spock: Exploiting Serverless Functions for SLO and Cost Aware Resource Procurement in Public Cloud. In *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*, pages 199–208. IEEE.
- Gunda, P. K., Ravindranath, L., Thekkath, C. A., Yu, Y., and Zhuang, L. (2010). Nectar: Automatic Management of Data and Computation in Datacenters. In *OSDI*, volume 10, pages 1–8.
- Gupta, A., Vanbever, L., Shahbaz, M., Donovan, S. P., Schlinker, B., Feamster, N., Rexford, J., Shenker, S., Clark, R., and Katz-Bassett, E. (2015). SDX: A Software Defined Internet Exchange. *ACM SIGCOMM Computer Communication Review*, 44(4):551–562.
- Gupta, A., Yang, F., Govig, J., Kirsch, A., Chan, K., Lai, K., Wu, S., Dhoot, S., Kumar, A., Agiwal, A., Bhansali, S., Hong, M., Cameron, J., Siddiqi, M., Jones, D., Shute, J., Gubarev, A., Venkataraman, S., and Agrawal, D. (2014). Mesa: Geo-Replicated, Near Real-Time, Scalable Data Warehousing. In *VLDB*.
- Gustafson, J. L. (1988). Reevaluating Amdahl’s law. *Communications of the ACM*, 31(5):532–533.
- Ha, S., Rhee, I., and Xu, L. (2008). CUBIC: A New TCP-Friendly High-Speed TCP Variant. *ACM SIGOPS Operating Systems Review*, 42(5):64–74.
- Hemminger, S. (2005). Network Emulation with NetEm. In *Linux Conf Au*, pages 18–23.
- Hindman, B., Konwinski, A., Zaharia, M., Ghodsi, A., Joseph, A. D., Katz, R. H., Shenker, S., and Stoica, I. (2011). Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center. In *NSDI*, volume 11, pages 22–22.
- Hong, C.-Y., Kandula, S., Mahajan, R., Zhang, M., Gill, V., Nanduri, M., and Wattenhofer, R. (2013). Achieving High Utilization with Software-Driven WAN. In *ACM SIGCOMM Computer Communication Review*, volume 43, pages 15–26. ACM.

- Hopps, C. (2000). Analysis of an Equal-Cost Multi-Path Algorithm. RFC 2992, RFC Editor.
- Hsieh, K., Harlap, A., Vijaykumar, N., Konomis, D., Ganger, G. R., Gibbons, P. B., and Mutlu, O. (2017). Gaia: Geo-Distributed Machine Learning Approaching $\text{\$}\{\text{\$lan}\}\text{\$}$ Speeds. In *14th $\text{\$}\{\text{\$USENIX}\}\text{\$}$ Symposium on Networked Systems Design and Implementation ($\text{\$}\{\text{\$NSDI}\}\text{\$}$ 17)*, pages 629–647.
- Hung, C.-C., Golubchik, L., and Yu, M. (2015). Scheduling jobs across geo-distributed datacenters. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*, pages 111–124. ACM.
- Hwang, J., Zeng, S., y Wu, F., and Wood, T. (2013). A Component-Based Performance Comparison of Four Hypervisors. In *2013 IFIP/IEEE International Symposium on Integrated Network Management (IM 2013)*, pages 269–276. IEEE.
- Hwang, J.-H., Cetintemel, U., and Zdonik, S. (2007). Fast and Reliable Stream Processing over Wide Area Networks. In *2007 IEEE 23rd International Conference on Data Engineering Workshop*, pages 604–613. IEEE.
- Isard, M., Budiu, M., Yu, Y., Birrell, A., and Fetterly, D. (2007). Dryad: Distributed Data-Parallel Programs From Sequential Building Blocks. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, pages 59–72.
- Isard, M., Prabhakaran, V., Currey, J., Wieder, U., Talwar, K., and Goldberg, A. (2009). Quincy: Fair Scheduling for Distributed Computing Clusters. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, pages 261–276.
- Jacobson, V. (1988). Congestion Avoidance and Control. *ACM SIGCOMM Computer Communication Review*, 18(4):314–329.
- Jain, S., Kumar, A., Mandal, S., Ong, J., Poutievski, L., Singh, A., Venkata, S., Wanderer, J., Zhou, J., Zhu, M., Zolla, J., Hözl, U., Stuart, S., and Vahdat, A. (2013). B4: Experience with a Globally-deployed Software Defined Wan. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM, SIGCOMM '13*, pages 3–14, New York, NY, USA. ACM.
- Jonathan, A., Chandra, A., and Weissman, J. (2016). Awan: Locality-aware resource manager for geo-distributed data-intensive applications. In *2016 IEEE International Conference on Cloud Engineering (IC2E)*, pages 32–41. IEEE.
- Kalnis, P., Ng, W. S., Ooi, B. C., Papadias, D., and Tan, K.-L. (2002). An Adaptive Peer-to-Peer Network for Distributed Caching of OLAP Results. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data, SIGMOD '02*, pages 25–36, New York, NY, USA. ACM.
- Kangasharju, J., Roberts, J., and Ross, K. W. (2002). Object Replication Strategies in Content Distribution Networks. *Computer Communications*, 25(4):376–383.
- Karp, R. M. (1972). Reducibility among Combinatorial Problems. In *Complexity of computer computations*, pages 85–103. Springer.
- Kivity, A., Kamay, Y., Laor, D., Lublin, U., and Liguori, A. (2007). kvm: the Linux Virtual Machine Monitor. In *In Proceedings of the 2007 Ottawa Linux Symposium (OLS'-07)*.
- Kloudas, K., Mamede, M., Preguiça, N., and Rodrigues, R. (2015). Pixida: optimizing data parallel jobs in wide-area data analytics. *Proceedings of the VLDB Endowment*, 9(2):72–83.

- Knauth, T. and Fetzer, C. (2012). Energy-Aware Scheduling for Infrastructure Clouds. In *4th IEEE International Conference on Cloud Computing Technology and Science Proceedings*, pages 58–65. IEEE.
- Koponen, T., Casado, M., Gude, N., Stribling, J., Poutievski, L., Zhu, M., Ramanathan, R., Iwata, Y., Inoue, H., Hama, T., et al. (2010). Onix: A Distributed Control Platform for Large-Scale Production Networks. In *OSDI*, volume 10, pages 1–6.
- Kraska, T., Pang, G., Franklin, M. J., Madden, S., and Fekete, A. (2013). MDCC: Multi-data center consistency. In *Proceedings of the 8th ACM European Conference on Computer Systems*, pages 113–126. ACM.
- Lakshman, T. and Madhow, U. (1997). The Performance of TCP/IP for Networks with High Bandwidth-Delay Products and Random Loss. *IEEE/ACM transactions on networking*, 5(3):336–350.
- Lamport, L. (1998). The Part-Time Parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169.
- Langley, A., Riddoch, A., Wilk, A., Vicente, A., Krasic, C., Zhang, D., Yang, F., Kouranov, F., Swett, I., Iyengar, J., et al. (2017). The QUIC Transport Protocol: Design and Internet-Scale Deployment. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pages 183–196.
- Lee, D., Choi, J., Kim, J. H., Noh, S. H., Min, S. L., Cho, Y., and Kim, C. S. (2001). LRFU: A Spectrum of Policies That Subsumes the Least Recently Used and Least Frequently Used Policies. *IEEE Trans. Comput.*, 50(12):1352–1361.
- Leith, D. and Shorten, R. (2004). H-TCP: TCP for High-Speed and Long-Distance Networks. In *Proceedings of PFLDnet*, volume 2004.
- Li, B., Li, J., Huai, J., Wo, T., Li, Q., and Zhong, L. (2009). Enacloud: An Energy-Saving Application Live Placement Approach for Cloud Computing Environments. In *2009 IEEE International Conference on Cloud Computing*, pages 17–24. IEEE.
- Li, H., Ghodsi, A., Zaharia, M., Shenker, S., and Stoica, I. (2014). Tachyon: Reliable, Memory Speed Storage for Cluster Computing Frameworks. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 1–15. ACM.
- Li, S., Da Xu, L., and Zhao, S. (2015). The internet of things: a survey. *Information Systems Frontiers*, 17(2):243–259.
- Li, Z., Duan, F., Nguyen, M., Che, H., Lei, Y., and Jiang, H. (2019). IPSO: A Scaling Model for Data-Intensive Applications. In *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*, pages 238–248. IEEE.
- Lillibridge, M., Eshghi, K., and Bhagwat, D. (2013). Improving Restore Speed for Backup Systems That Use Inline chunk-based Deduplication. In *11th USENIX Conference on File and Storage Technologies (FAST 13)*, pages 183–197.
- Lim, S.-H., Ko, Y.-B., Jung, G.-H., Kim, J., and Jang, M.-W. (2014). Inter-Chunk Popularity-Based Edge-First Caching in Content-Centric Networking. *IEEE Communications Letters*, 18(8):1331–1334.
- Litzkow, M. J., Livny, M., and Mutka, M. W. (1987). Condor - a Hunter of Idle Workstations. Technical report, University of Wisconsin-Madison Department of Computer Sciences.

- Logothetis, D., Olston, C., Reed, B., Webb, K., and Yocum, K. (2010). Stateful Bulk Processing for Incremental Algorithms. In *ACM Symposium on Cloud Computing (SOCC)*, volume 10.
- Magnusson, P. S., Larsson, F., Moestedt, A., Werner, B., Nilsson, J., Stenström, P., Lundholm, F., Karlsson, M., Dahlgren, F., and Grahm, H. (1998). SimICS/Sun4m: A Virtual Workstation. In *USENIX Annual Technical Conference*, pages 119–130.
- Malewicz, G., Austern, M. H., Bik, A. J., Dehnert, J. C., Horn, I., Leiser, N., and Czajkowski, G. (2010). Pregel: a System for Large-Scale Graph Processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, pages 135–146.
- Mao, M. and Humphrey, M. (2011). Auto-Scaling to Minimize Cost and Meet Application Deadlines in Cloud Workflows. In *SC'11: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–12. IEEE.
- Mao, M., Li, J., and Humphrey, M. (2010). Cloud Auto-Scaling with Deadline and Budget Constraints. In *2010 11th IEEE/ACM International Conference on Grid Computing*, pages 41–48. IEEE.
- Mascolo, S., Casetti, C., Gerla, M., Sanadidi, M. Y., and Wang, R. (2001). TCP Westwood: Bandwidth Estimation for Enhanced Transport over Wireless Links. In *Proceedings of the 7th Annual International Conference on Mobile Computing and Networking*, pages 287–297.
- Mathis, M., Semke, J., Mahdavi, J., and Ott, T. (1997). The macroscopic behavior of the TCP congestion avoidance algorithm. *ACM SIGCOMM Computer Communication Review*, 27(3):67–82.
- McKeown, N., Anderson, T., Balakrishnan, H., Parulkar, G., Peterson, L., Rexford, J., Shenker, S., and Turner, J. (2008). OpenFlow: Enabling Innovation in Campus Networks. *ACM SIGCOMM Computer Communication Review*, 38(2):69–74.
- McPherson, D. and Dykes, B. (2001). VLAN Aggregation for Efficient IP Address Allocation. RFC 3069, RFC Editor.
- Meyer, M. and Vasseur, J. (2010). MPLS traffic engineering soft preemption.
- Ming, Z., Xu, M., and Wang, D. (2014). Age-Based Cooperative Caching in Information-Centric Networking. In *Computer Communication and Networks (ICCCN), 2014 23rd International Conference on*, pages 1–8. IEEE.
- Mohan, P., Thakurta, A., Shi, E., Song, D., and Culler, D. (2012). GUPT: Privacy Preserving Data Analysis Made Easy. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 349–360.
- Muralidhar, S., Lloyd, W., Roy, S., Hill, C., Lin, E., Liu, W., Pan, S., Shankar, S., Sivakumar, V., Tang, L., et al. (2014). f4: Facebook’s Warm BLOB Storage System. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 383–398.
- Nace, D., Doan, N.-L., Gourdin, E., and Liau, B. (2006). Computing optimal max-min fair resource allocation for elastic flows. *IEEE/ACM Transactions on Networking*, 14(6):1272–1281.
- Nygren, E., Sitaraman, R. K., and Sun, J. (2010). The Akamai Network: A Platform for High-Performance Internet Applications. *ACM SIGOPS Operating Systems Review*, 44(3):2–19.
- Olston, C., Reed, B., Silberstein, A., and Srivastava, U. (2008). Automatic Optimization of Parallel Dataflow Programs. In *USENIX Annual Technical Conference*, pages 267–273.

- Ongaro, D. and Ousterhout, J. (2014). In Search of an Understandable Consensus Algorithm. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 305–319.
- Padhye, J., Firoiu, V., Towsley, D., and Kurose, J. (1998). Modeling TCP Throughput: A Simple Model and Its Empirical Validation. In *Proceedings of the ACM SIGCOMM'98 conference on Applications, technologies, architectures, and protocols for computer communication*, pages 303–314.
- Pahl, C., Brogi, A., Soldani, J., and Jamshidi, P. (2017). Cloud Container Technologies: a State-of-the-Art Review. *IEEE Transactions on Cloud Computing*.
- Panigrahy, R., Talwar, K., Uyeda, L., and Wieder, U. (2011). Heuristics for vector bin packing. *research.microsoft.com*.
- Peng, B., Hosseini, M., Hong, Z., Farivar, R., and Campbell, R. (2015). R-storm: Resource-aware scheduling in storm. In *Proceedings of the 16th Annual Middleware Conference*, pages 149–161. ACM.
- Pfaff, B., Pettit, J., Amidon, K., Casado, M., Koponen, T., and Shenker, S. (2009). Extending Networking into the Virtualization Layer. In *Hotnets*.
- Pfaff, B., Pettit, J., Koponen, T., Jackson, E., Zhou, A., Rajahalme, J., Gross, J., Wang, A., Stringer, J., Shelar, P., et al. (2015). The Design and Implementation of Open vSwitch. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 117–130.
- Pietzuch, P., Ledlie, J., Shneidman, J., Roussopoulos, M., Welsh, M., and Seltzer, M. (2006). Network-Aware Operator Placement for Stream-Processing Systems. In *22nd International Conference on Data Engineering (ICDE'06)*, pages 49–49. IEEE.
- Popa, L., Budiu, M., Yu, Y., and Isard, M. (2009). DryadInc: Reusing Work in Large-Scale Computations. In *Proceedings of the 2009 Conference on Hot Topics in Cloud Computing, HotCloud'09*, Berkeley, CA, USA. USENIX Association.
- Pu, Q., Ananthanarayanan, G., Bodik, P., Kandula, S., Akella, A., Bahl, P., and Stoica, I. (2015). Low Latency Geo-distributed Data Analytics. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication, SIGCOMM '15*, pages 421–434, New York, NY, USA. ACM.
- Pucher, A., Wolski, R., and Krintz, C. (2017). EXFed: Efficient Cross-Federation with Availability SLAs on Preemptible IaaS Instances. In *2017 IEEE International Conference on Cloud Engineering (IC2E)*, pages 75–80. IEEE.
- Qu, C., Calheiros, R. N., and Buyya, R. (2016). A Reliable and Cost-Efficient Auto-Scaling System for Web Applications Using Heterogeneous Spot Instances. *Journal of Network and Computer Applications*, 65:167–180.
- Rabkin, A., Arye, M., Sen, S., Pai, V. S., and Freedman, M. J. (2014). Aggregation and Degradation in JetStream: Streaming Analytics in the Wide Area. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation, NSDI'14*, pages 275–288, Berkeley, CA, USA. USENIX Association.
- Raiciu, C., Paasch, C., Barre, S., Ford, A., Honda, M., Duchene, F., Bonaventure, O., and Handley, M. (2012). How Hard Can It Be? Designing and Implementing a Deployable Multipath TCP. In *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, pages 399–412.

- Rajasekar, A., Moore, R., Hou, C.-y., Lee, C. A., Marciano, R., de Torcy, A., Wan, M., Schroeder, W., Chen, S.-Y., Gilbert, L., and others (2010). iRODS Primer: integrated rule-oriented data system. *Synthesis Lectures on Information Concepts, Retrieval, and Services*, 2(1):1–143.
- Raman, R., Livny, M., and Solomon, M. (1999). Matchmaking: An Extensible Framework for Distributed Resource Management. *Cluster Computing*, 2(2):129–138.
- Reiss, C., Tumanov, A., Ganger, G. R., Katz, R. H., and Kozuch, M. A. (2012). Heterogeneity and Dynamicity of Clouds at Scale: Google Trace Analysis. In *Proceedings of the Third ACM Symposium on Cloud Computing*, page 7. ACM.
- Rivest, R. (1992). The MD5 Message-Digest Algorithm. RFC 1312, RFC Editor.
- Rizzo, L. and Vicisano, L. (2000). Replacement Policies for a Proxy Cache. *IEEE/ACM Trans. Netw.*, 8(2):158–170.
- Sandberg, R., Goldberg, D., Kleiman, S., Walsh, D., and Lyon, B. (1985). Design and Implementation of the Sun Network Filesystem. In *Proceedings of the Summer USENIX conference*, pages 119–130.
- Sermersheim, J. (2006). Lightweight Directory Access Protocol (LDAP): The Protocol. RFC 4511, RFC Editor.
- Shafiq, M. Z., Liu, A. X., and Khakpour, A. R. (2014). Revisiting Caching in Content Delivery Networks. In *the 2014 ACM International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '14, pages 567–568, New York, NY, USA. ACM.
- Shah, T., Rabhi, F., and Ray, P. (2015). Investigating an Ontology-Based Approach for Big Data Analysis of Inter-Dependent Medical and Oral Health Conditions. *Cluster Computing*, 18(1):351–367.
- Shvachko, K., Kuang, H., Radia, S., and Chansler, R. (2010). The Hadoop Distributed File System. In *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–10. IEEE.
- Sindelar, M., Sitaraman, R. K., and Shenoy, P. (2011). Sharing-Aware Algorithms for Virtual Machine Colocation. In *Proceedings of the Twenty-Third Annual ACM Symposium on Parallelism in Algorithms and Architectures*, pages 367–378.
- Singh, A., Ong, J., Agarwal, A., Anderson, G., Armistead, A., Bannon, R., Boving, S., Desai, G., Felderman, B., Germano, P., Kanagala, A., Provost, J., Simmons, J., Tanda, E., Wanderer, J., Hölzle, U., Stuart, S., and Vahdat, A. (2015). Jupiter Rising: A Decade of Clos Topologies and Centralized Control in Google’s Datacenter Network. In *Sigcomm '15*.
- Sivarajah, U., Kamal, M. M., Irani, Z., and Weerakkody, V. (2017). Critical Analysis of Big Data Challenges and Analytical Methods. *Journal of Business Research*, 70:263–286.
- Sourlas, V., Gkatzikis, L., Flegkas, P., and Tassioulas, L. (2013). Distributed Cache Management in Information-Centric Networks. *IEEE Transactions on Network and Service Management*, 10(3):286–299.
- Staples, G. (2006). TORQUE Resource Manager. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, pages 8–es.
- Stefanov, E. and Shi, E. (2013). Multi-cloud Oblivious Storage. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security, CCS '13*, pages 247–258, New York, NY, USA. ACM.

- Sun, X.-H. and Chen, Y. (2010). Reevaluating Amdahl’s Law in the Multicore Era. *Journal of Parallel and distributed Computing*, 70(2):183–188.
- Tordini, F., Aldinucci, M., Viviani, P., Ivan, M., Liò, P., et al. (2018). Scientific Workflows on Clouds with Heterogeneous and Preemptible Instances. In *International Conference on Parallel Computing (ParCo 2017)*, pages 1–10. IOS Press.
- Uhlig, R., Neiger, G., Rodgers, D., Santoni, A. L., Martins, F. C., Anderson, A. V., Bennett, S. M., Kagi, A., Leung, F. H., and Smith, L. (2005). Intel Virtualization Technology. *Computer*, 38(5):48–56.
- Van den Bossche, R., Vanmechelen, K., and Broeckhove, J. (2010). Cost-Optimal Scheduling in Hybrid IaaS Clouds for Deadline Constrained Workloads. In *2010 IEEE 3rd International Conference on Cloud Computing*, pages 228–235. IEEE.
- Van der Ham, J. J., Dijkstra, F., Travostino, F., Andree, H. M., and de Laat, C. T. (2006). Using RDF to Describe Networks. *Future Generation Computer Systems*, 22(8):862–867.
- Varghese, B. and Buyya, R. (2018). Next Generation Cloud Computing: New Trends and Research Directions. *Future Generation Computer Systems*, 79:849–861.
- Viswanathan, R., Ananthanarayanan, G., and Akella, A. (2016). CLARINET: WAN-Aware Optimization for Analytics Queries. In *OSDI*, volume 16, pages 435–450.
- Vulimiri, A., Curino, C., Godfrey, P. B., Jungblut, T., Karanasos, K., Padhye, J., and Varghese, G. (2015a). WANalytics: Geo-Distributed Analytics for a Data Intensive World. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD ’15, pages 1087–1092, New York, NY, USA. ACM. event-place: Melbourne, Victoria, Australia.
- Vulimiri, A., Curino, C., Godfrey, P. B., Jungblut, T., Padhye, J., and Varghese, G. (2015b). Global Analytics in the Face of Bandwidth and Regulatory Constraints. pages 323–336.
- Weil, S. A., Brandt, S. A., Miller, E. L., Long, D. D., and Maltzahn, C. (2006a). Ceph: A scalable, high-performance distributed file system. In *Proceedings of the 7th symposium on Operating systems design and implementation*, pages 307–320.
- Weil, S. A., Brandt, S. A., Miller, E. L., and Maltzahn, C. (2006b). CRUSH: Controlled, scalable, decentralized placement of replicated data. In *SC’06: Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, pages 31–31. IEEE.
- Whitaker, A., Shaw, M., Gribble, S. D., et al. (2002). Denali: Lightweight Virtual Machines for Distributed and Networked Applications. Technical report, Technical Report 02-02-01, University of Washington.
- Wu, Z., Butkiewicz, M., Perkins, D., Katz-Bassett, E., and Madhyastha, H. V. (2013). Spanstore: Cost-Effective Geo-Replicated Storage Spanning Multiple Cloud Services. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 292–308.
- Wu, Z. and Madhyastha, H. V. (2013). Understanding the Latency Benefits of Multi-Cloud Webservice Deployments. *ACM SIGCOMM Computer Communication Review*, 43(2):13–20.
- Wu, Z., Yu, C., and Madhyastha, H. V. (2015). CostTLO: Cost-Effective Redundancy for Lower Latency Variance on Cloud Storage Services. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 543–557.

- Xu, J., Chen, Z., Tang, J., and Su, S. (2014). T-Storm: Traffic-Aware Online Scheduling in Storm. In *2014 IEEE 34th International Conference on Distributed Computing Systems*, pages 535–544.
- Xu, L., Harfoush, K., and Rhee, I. (2004). Binary Increase Congestion Control (BIC) for Fast Long-Distance Networks. In *IEEE INFOCOM 2004*, volume 4, pages 2514–2524. IEEE.
- Zaharia, M., Borthakur, D., Sen Sarma, J., Elmeleegy, K., Shenker, S., and Stoica, I. (2010a). Delay Scheduling: a Simple Technique for Achieving Locality and Fairness in Cluster Scheduling. In *Proceedings of the 5th European conference on Computer systems*, pages 265–278.
- Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauley, M., Franklin, M. J., Shenker, S., and Stoica, I. (2012). Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI’12, pages 2–2, Berkeley, CA, USA. USENIX Association.
- Zaharia, M., Chowdhury, M., Franklin, M. J., Shenker, S., Stoica, I., et al. (2010b). Spark: Cluster Computing with Working Sets. *HotCloud*, 10(10-10):95.
- Zhang, G., Li, Y., and Lin, T. (2013). Caching in Information Centric Networking: A Survey. *Computer Networks*, 57(16):3128–3141.
- Zhang, Q., Cheng, L., and Boutaba, R. (2010). Cloud Computing: State-of-the-Art and Research Challenges. *Journal of Internet Services and Applications*, 1(1):7–18.
- Zheng, C., Rupprecht, L., Tarasov, V., Thain, D., Mohamed, M., Skourtis, D., Warke, A. S., and Hildebrand, D. (2018). Wharf: Sharing Docker Images in a Distributed File System. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 174–185.
- Zhou, S. (1992). LSF: Load Sharing in Large Heterogeneous Distributed Systems. In *In Workshop on Cluster Computing*, volume 136.
- Zipf, G. K. (1929). Relative Frequency as a Determinant of Phonetic Change. *Harvard Studies in Classical Philology*, 40:1–95.